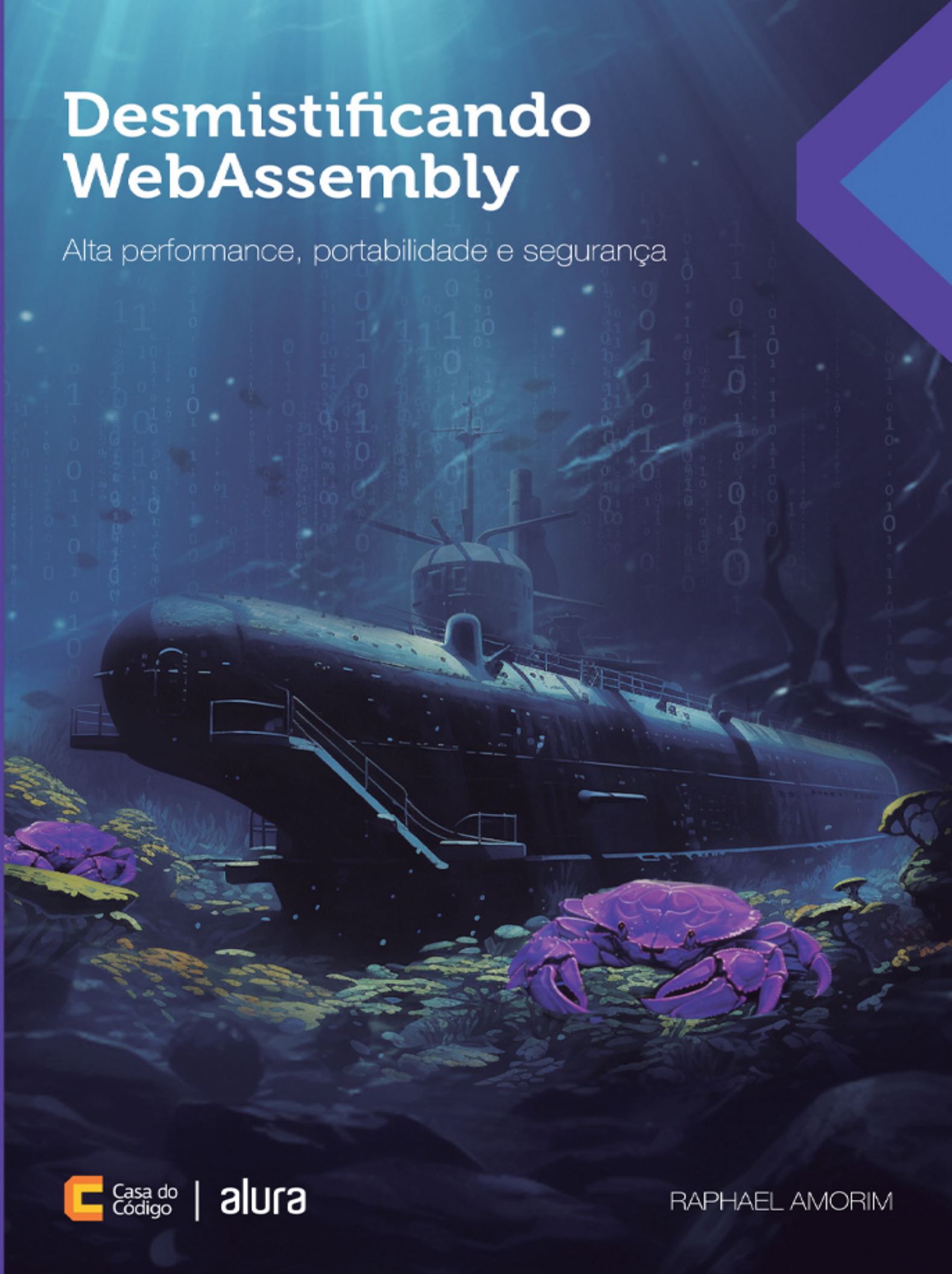
Desmistificando WebAssembly

Alta performance, portabilidade e segurança



Casa do Código | alura

RAPHAEL AMORIM

Sumário

- ISBN
- Prefácio
- Sobre o autor
- Dedicatória
- Agradecimentos
- 1. Introdução
- 2. Fundamentos ao primeiro módulo
- 3. Por trás da magia no navegador
- 4. Criando um editor de imagem com WebAssembly
- 5. Vinculação e interface binária de aplicativo
- 6. Encoding e Strings
- 7. Threads e atômicos
- 8. Tipos de erro
- 9. Destrinchando o formato de texto
- 10. Estruturas binárias, WABT e Binaryen
- 11. WASI - WebAssembly System Interface
- 12. Referências bibliográficas

ISBN

Impresso: 978-85-5519-346-0

Digital: 978-85-5519-347-7

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Amorim, Raphael
Desmistificando webassembly : alta performance, portabilidade e segurança / Raphael Amorim. -- São Paulo : AOVS Sistemas de Informática, 2023.

ISBN 978-85-5519-346-0

1. Ciência da Computação 2. Linguagem de programação 3. Performance 4. WEB (Linguagem de programação) I. Título.

23-168033 CDD-005.133

Índices para catálogo sistemático:

1. WEB : Linguagem de programação : Computadores : Processamento de dados 005.133

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Prefácio

O WebAssembly, também conhecido como WASM, é um formato de código dedicado e otimizado para a Web, que pode melhorar muito o desempenho dos sites. Apesar de ter sido criado originalmente para servir à Web, ele não se limita apenas a ela e vem sendo usado em diferentes lugares e propósitos, como dispositivos móveis e ambientes de computação de borda (do inglês, *Edge computing*), tais como os *Workers* da Cloudflare e outros.

O arquivo binário do WASM pode ser produzido como destino de compilação de inúmeras linguagens de programação como Rust, C, C++, Golang, Lua, Zig, Python, Swift, Ruby, PHP, Java, C# e tantas outras. Esse produto contém instruções de baixo nível para ser executável com uma velocidade de alta performance, muitas vezes próxima da velocidade nativa.

Além de tornar a Web ainda mais rápida, WebAssembly adiciona pluralidade, pois viabiliza que pessoas programadoras de diversas linguagens criem aplicações ou portabilizem aplicações existentes de plataformas nativas para os navegadores. As aplicações construídas com o WASM possuem diferentes características de segurança em comparação com qualquer código escrito em JavaScript e executado no navegador.

A segurança na Web foi revisitada e empoderada na construção do WebAssembly. O modelo de segurança do WASM foi projetado para ser completamente isolado em "caixa de areia" (*sandbox*), e isso significa que o WASM não disponibiliza acesso nem se comunica com nenhum contexto externo por padrão. Essa comunicação precisa ser feita de forma declarativa pelo(a) programador(a).

No decorrer do livro, vamos ver mais sobre as caraterísticas da segurança, performance, portabilidade e pluralidade de linguagens. Este livro tem como objetivo fazer a apresentação do WebAssembly de maneira prática com foco no uso da tecnologia no dia a dia, até o completo entendimento de como funciona uma estrutura de arquivo

binário e todas as suas instruções. Com esse conhecimento, você conseguirá até mesmo escrever seu próprio compilador WebAssembly.

O material foi pensado para pessoas desenvolvedoras que desejam entender como o WebAssembly funciona a fundo. Vamos examinar e dissecar desde o processo de compilação de módulo até a sua execução com a máquina virtual do WASM. Ao longo dos capítulos, colocaremos ênfase e exemplificaremos inúmeros conceitos da ciência da computação com o ecossistema do WebAssembly.

Existem diferentes perfis de pessoas programadoras que poderão tirar proveito do conteúdo deste livro:

- Programadores(as) *front-end* que desejam criar aplicações para a Web com uma performance próxima da nativa;
- Programadores(as) *back-end* que desejam portar aplicações escritas originalmente para plataformas nativas para a Web;
- Programadores(as) que desejam criar aplicações de característica multiplataforma, ou seja, executar o mesmo código para plataformas distintas;
- Programadores(as) que desejam aprender como depurar ou otimizar arquivos binários WebAssembly;
- Qualquer programador(a) que deseja entender a fundo como o WebAssembly funciona.

A maior parte dos códigos dos exemplos foi escrita com as linguagens de programação JavaScript e Rust. Eventualmente, ao decorrer do livro, ambas as linguagens serão menos utilizadas, pois aprenderemos e usaremos a escrita de módulos WebAssembly com sua representação textual.

O nível de conhecimento necessário sobre JavaScript como linguagem de programação é o intermediário, e isso significa ter experiência prática como programador(a) na escrita de aplicações para a Web ou em serviços escritos em NodeJS. É requerido ter a

familiaridade básica com a linguagem, como sua sintaxe, variáveis, funções, laços de repetição, escopo léxico, *closures* e outros.

O conhecimento necessário sobre Rust como linguagem de programação é entre o básico e o intermediário. Todas as ferramentas do ecossistema da linguagem serão introduzidas e não é esperado que o leitor ou a leitora tenha experiência na escrita de aplicações em Rust. Entretanto, é necessário que possua o entendimento da sintaxe da linguagem, além de entender como funcionam funções, tipos, laços de repetição, referência e *borrowing*.

Versões

A versão do Rust utilizada oficialmente neste exemplar é a versão 1.70.0. Os exemplos de código também funcionam em versões anteriores, desde a versão 1.50.0 em diante. É bem provável que também funcione sem demais problemas para futuras versões que não contêm mudanças drásticas no compilador da linguagem, em razão da simplicidade dos exemplos.

Vamos focar majoritariamente no estudo da versão 1.0 do WebAssembly (com pequenas exceções nos últimos capítulos). O WebAssembly 1.0 é considerado o conjunto essencial de funcionalidades e recursos e possui o suporte habilitado na maioria dos navegadores atuais.

Este material não tem como objetivo ensinar ferramentas específicas do universo da linguagem Rust para trabalhar com WebAssembly; pelo contrário, o objetivo é aprender os fundamentos da tecnologia para que a pessoa desenvolvedora não se torne dependente de uma linguagem ou utilitários específicos para trabalhar com WebAssembly. Nos últimos capítulos, escreveremos a lógica dos módulos com o uso da representação textual do WebAssembly em vez de Rust.

Estrutura do livro

O livro é divido em três etapas:

Na primeira, vamos nos debruçar sobre a história, objetivos, fundamentos e uso prático do WebAssembly em aplicações para a Web. Ainda na primeira etapa, escreveremos uma aplicação Web que realiza processamento de imagens com o uso de funções escritas originalmente em Rust e compilada posteriormente para WebAssembly.

A segunda etapa é dedicada a conceitos computacionais necessários para a escrita de aplicações mais complexas no WebAssembly, como codificação de dados, vinculação dinâmica e estática, instruções atômicas, memória compartilhada e outros. Veremos as aplicações práticas de cada conceito com exemplos de módulos WebAssembly.

A última etapa é a união das duas primeiras partes para ver como o coração do WebAssembly funciona, como os tipos de erro, depuração de binários, máquina virtual, conjunto de instruções, estrutura do formato binário e outros. Em especial, o último capítulo do livro detalha sobre diferentes tipos de hospedeiros e como usar até mesmo WebAssembly no *back-end* com característica multiplataforma e uso de interface nativa de sistema operacional.

Este livro pesa e enfatiza o aprendizado dos fundamentos do WebAssembly do começo ao fim. Ao final do livro, você terá todo o conhecimento necessário para não depender de linguagem, ferramenta ou recurso específico além do próprio WebAssembly para desenvolver aplicações, seja para a Web ou para qualquer outro destino de uso.

Sobre o autor

Hugo Raphael Vianna Amorim nasceu no Rio de Janeiro, passou a maior parte de sua infância em uma cidade chamada Itaboraí e, hoje em dia, vive em Estocolmo, na Suécia. Começou a programar com 15 anos por diversão e não parou nunca mais. Formado em análise e desenvolvimento de sistemas pelo Instituto Infnet, foi palestrante em múltiplas conferências nacionais e internacionais.

Atualmente, trabalha como Sr. Software Engineer no Viaplay, tendo também trabalhado em empresas como Spotify, GoDaddy e Globo.com nos últimos 10 anos. Raphael foi responsável pela implementação e adoção da linguagem de programação Rust, e eventualmente WebAssembly, na maior empresa de *streaming* de vídeos dos países nórdicos.

Apaixonado por Open Source desde muito cedo, ele se envolveu com diversos projetos abertos, especialmente no cenário de desenvolvimento Web, sendo contribuidor do pacote binário oficial do WebAssembly, jQuery, React, entre outros. Além de atuar como contribuidor, mantém o emulador de terminal conhecido como "Rio" e o React-TV. Nas horas vagas, ele tem como lazer a corrida e a pintura a óleo.

Dedicatória

Durante toda a minha vida, eu sempre fui um leitor ávido. Filho de dois professores, ambos me incentivaram a desenvolver a paixão pela leitura. Minha mãe (professora de português e espanhol) e meu pai (professor de biologia e química) sempre nos presenteavam (minha irmã e eu) com livros, e isso tornou-se um hábito dentro de nossa casa.

Uma das memórias mais vívidas que tenho quando penso em minha infância são as excursões para a "Bienal do Livro" no Rio de Janeiro. Eu juntava dinheiro no ano do evento e quebrava o meu "cofre de porquinho" para poder comprar livros, mangás e revistas em quadrinhos.

Dedico este livro a minha mãe e ao meu saudoso pai, que sempre me incentivaram a ler e escrever, emergindo em um mundo totalmente diferente de tudo que vivi.

Agradecimentos

O conhecimento usado neste trabalho não é derivado de um homem só e, por esse motivo, é necessário fazer um preito de gratidão àqueles que colaboraram com minhas pesquisas e estudos.

Primeiramente, agradeço a Deus pela existência de elementos que me inspiram constantemente, como o mar, a música e o amor.

Agradeço a minha família e amigos, que me incentivaram a escrever este livro. Em especial, ao meu pai, que já se foi, mas continua sendo minha maior inspiração.

Agradecimentos à comunidade de desenvolvimento do WebAssembly, com destaque em especial aos membros do comitê que se dispuseram a responder minhas inúmeras perguntas e compartilharam seu precioso tempo para me auxiliar com as pesquisas deste exemplar.

Agradecimentos à Vivian Matsui, responsável por inúmeras revisões e comentários importantes. Sem suas valiosas contribuições, não seria possível concretizar esta obra.

Capítulo 1
Introdução

Bem-vindo(a) ao fantástico mundo do WebAssembly!

Quantas vezes você já não ouviu alguém falar sobre *WebAssembly* (ou WASM, abreviadamente) em conferências, palestras, no escritório do trabalho ou até mesmo em um artigo de blog ou revista de tecnologia? Se você trabalha com programação, suponho que você já tenha passado por isso, uma vez que grandes empresas como Google, Microsoft, entre outras, estão continuamente criando experimentos e projetos com WebAssembly.

Ordinariamente, tais menções costumam dar destaque à performance ou à segurança. Porém, infelizmente, o estado atual do uso da tecnologia ainda é visto pela maioria de desenvolvedores e desenvolvedoras Web como um "bicho de sete cabeças". Existe um sentimento comum de receio sobre o desenvolvimento com WebAssembly e diversos fatores justificam essa reação na comunidade Web, alguns dos quais são:

- WebAssembly é frequentemente apresentado com fortes conceitos aplicados de Ciência da Computação, como codificação de dados (*data encoding*, em inglês). Tais conceitos dificultam o aprendizado para desenvolvedores inexperientes;
- Falta de diversificação dos exemplos de uso na internet, com destaque à alta complexidade dos exemplos existentes;
- Os materiais de aprendizado são majoritariamente em inglês;
- Comparativos de velocidade de execução entre JavaScript e WebAssembly para operações que o WebAssembly não foi criado para resolver, resultando em uma complexidade maior para resolver problemas simples.

Esses são fatores que pude ver aplicados na prática do meu dia a dia quando decidi experimentar e adotar o uso do WebAssembly para

evoluir a performance e a segurança de aplicações Web em algumas empresas em que trabalhei. Até os dias atuais, não me esqueço da reação de inúmeros colegas de trabalho que arregalaram os olhos quando apresentei o WASM pelo simples fato de considerar tal tecnologia algo mitificado.

O objetivo deste livro é justamente desmistificar o WebAssembly e apresentar você a essa tecnologia amplamente utilizada por diversas empresas, produtos e pesquisas.

Antes de entrar de cabeça nessa jornada, precisamos entender quais tipos de problemas o WebAssembly foi criado para resolver. E para responder a essa pergunta, é preciso olhar o estado de desenvolvimento que precedeu o WebAssembly. Vamos então relembrar alguns eventos importantes no cenário do desenvolvimento Web de forma cronológica.

1.1 Do NodeJS ao WebAssembly

Faça comigo um pequeno exercício mental de contextualização e imagine os seguintes eventos: o ano é 2009 e o NodeJS acaba de ser lançado, mudando drasticamente o futuro do desenvolvimento de software nos anos seguintes. De 2010 até 2012, o ecossistema passou por importantes evoluções: ferramentas e frameworks como *npm*, *express* e *socket.io* foram criados e, nesse meio período, empresas importantes como LinkedIn e Uber adotaram o uso do NodeJS.

Seria correto afirmar que, em 2012, o JavaScript estava em alta e sendo usado de forma abrangente, não apenas no cenário front-end, mas também no back-end. Um ano depois, especificamente em maio de 2013, o Facebook e Jordan Walker anunciavam o que seria uma

das bibliotecas mais utilizadas nos próximos anos: o *React.js*, trazendo consigo o React Native alguns anos depois, que desencadearia conversas sobre desenvolvimento JavaScript no âmbito mobile.

No mesmo ano, o *asm.js* foi criado e basicamente consistia em um *subset* do JavaScript onde você podia escrever código em linguagens de tipagem estática com gerenciamento manual de memória como C e C++, além de produzir código JavaScript em uma velocidade na execução do código próxima ao nativo.

Em 2014, tivemos o lançamento oficial do HTML5 pela W3C (lembrando que o primeiro *draft release* foi em 2008 pelo WHATWG, *Web Hypertext Application Technology Working Group*) e, no ano seguinte, em 2015, o WebAssembly foi anunciado.

O WebAssembly foi primeiramente anunciado em uma demonstração que portava e executava *Unity Angry Bots* no Firefox, Chrome e Microsoft Edge. Desde sua demonstração, não tardou a fazer sua aparição oficial nos navegadores...

1.2 O que é WebAssembly?

Figura 1.1: Logo do WebAssembly.

Em junho de 2017, Brendan Eich, criador do JavaScript e cofundador do Mozilla Project, da Mozilla Foundation e da Mozilla Corporation, anunciou que a plataforma Web iria receber um novo formato binário de baixo nível que faria um trabalho mais eficaz e performático como um executável do que o próprio JavaScript. A especificação do formato de código binário para navegadores foi nomeada então como WebAssembly.

Além da apresentação do WebAssembly, destinado a ser mais rápido de analisar (*parse*) pelo navegador do que o asm.js, também foi revelado que empresas como Google, Microsoft, Mozilla, Apple e outras estariam trabalhando secretamente de forma conjunta em um projeto por meio de um grupo (também conhecido como *W3C WebAssembly Community Group*) interessado na criação e desenvolvimento do WASM.

No mesmo ano, o navegador da Apple, o Safari, em sua 11ª versão, adicionou suporte ao WASM e se juntou ao Chrome, Firefox e Opera, que já possuíam WebAssembly habilitado por padrão. O navegador Microsoft Edge até então também tinha o suporte inicial ao WASM, porém precisava ser habilitado manualmente pelo usuário.

O WebAssembly define um formato de código binário portátil e corresponde ao formato de texto para programas executáveis, tendo consigo dois formatos: `wat` (formato de texto) e `wasm` (formato binário). O objetivo do WASM é tornar possíveis aplicações de altíssima performance em páginas Web, mas também sendo agnóstico à plataforma. Melhor dizendo: não há funcionalidades específicas para a Web no WASM. Isso significa que pode ser executado em qualquer outra plataforma.

Sendo um padrão aberto, também tem como objetivo ter suporte para diversas linguagens e sistemas operacionais. Muitas linguagens populares adotaram suporte em pelo menos algum nível. Você não precisa saber como criar código WebAssembly para usá-lo. Os módulos do WebAssembly podem ser importados para um aplicativo Web (ou Node.js), expondo as funções do WebAssembly para uso

via JavaScript. Outro importante ponto é que o WASM foi especificado para ser executado em um ambiente seguro e com permissões bem restritas. Como qualquer outro código da Web, ele aplica a política do navegador de mesma origem e permissões atribuídas.

É certo afirmar que o asm.js influenciou o WASM de múltiplas formas. Diversos problemas existentes no asm.js eventualmente se tornaram conceitos-chaves por trás do WebAssembly. Alguns dos maiores problemas do asm.js eram:

- Sua execução acontecia em navegadores e o desempenho dependia muito do hardware e do motor do navegador (V8, SpiderMonkey etc.);
- A segurança e portabilidade do asm.js, especialmente pela facilidade de obtenção de dados importantes pelo depurador do JavaScript em diversos navegadores. Esse problema era amplamente discutido para evitar a exposição de lógicas sensíveis de aplicações já escritas em outras linguagens;
- Dificuldade na leitura do código produzido pelo compilador asm.js.

Todos esses tipos de problemas tiveram um peso na influência do desenvolvimento do WebAssembly no decorrer dos anos. Quando o WASM foi lançado, já havia uma bagagem de anos de discussão de problemas recorrentes de seus precursores, e isso tornou possível a rápida adoção da tecnologia em vários casos de uso.

Código de baixo nível para a Web

Diversas áreas se beneficiam da existência do WASM. Alguns dos maiores beneficiados no uso desse formato de código binário são as *threads* e *SIMD* (*Single Instruction Multiple Data*). A ideia por trás do SIMD é bem simples: basicamente, é ter vários pequenos pedaços de dados e executar uma função para processar todos esses pequenos pedaços. Você provavelmente já consegue ver onde isso pode ser aplicado, né?

Um dos casos de uso que provavelmente vem à cabeça é o processamento paralelo de *streams* de vídeo. Esse é apenas um dos muitos casos onde que você simplesmente quer esquecer dos recursos da linguagem dinâmica, como o coletor de lixo (em inglês, *Garbage Collector*), sistema de tipos por objeto e outros.

Você quer processar da forma mais otimizada possível, e isso significa trazer um programa baseado em um código binário otimizado que foi feito exclusivamente para lidar com esse tipo de operação. A mesma tecnologia abre muitas possibilidades em diversas áreas, como jogos, realidade virtual, realidade aumentada e tantos outros exemplos.

Códigos binários têm como vantagem a possibilidade de criar pacotes de aplicativos menores do que com JavaScript; entretanto, é bem comum ser inviável a leitura do código de tais formatos por seres humanos. Provavelmente você está se perguntando: *como faremos para depurar um formato de idioma binário?*

Quando você trabalha com a Web, acaba lidando muito com o depurador do navegador, e é nesse mesmo recurso que o navegador provê uma representação simplificada da estrutura semântica, também conhecida como árvore sintática abstrata (do inglês, *Abstract Syntax Tree*), que é representada em um formato de texto de forma (moderadamente) amigável. Vamos entrar em exemplos no decorrer do livro.

Esse formato de texto inegavelmente será um pouco menos legível do que o código JavaScript, porém mais legível que o asm.js compilado.

É normal ver aplicativos Web usando WebAssembly em casos em que existem grandes transmissões de dados por meio de redes de funções de processamento, como um efeito de filtro em um vídeo de tempo real, aplicações em que tradicionalmente a maioria das pessoas pensa ser loucura usar JavaScript.

O JavaScript é uma ótima linguagem para construir a maior parte do código exigido por qualquer aplicativo que você possa imaginar, mas existe uma necessidade e motivo da existência do WebAssembly: fornecer o poder do código de "baixo nível" para a Web. Escrever um código feito em Rust, Go, C++ e outras linguagens e ser capaz de criar e empacotar um formato de código binário para Web é algo que abre portas para muitas ideias e possibilidades.

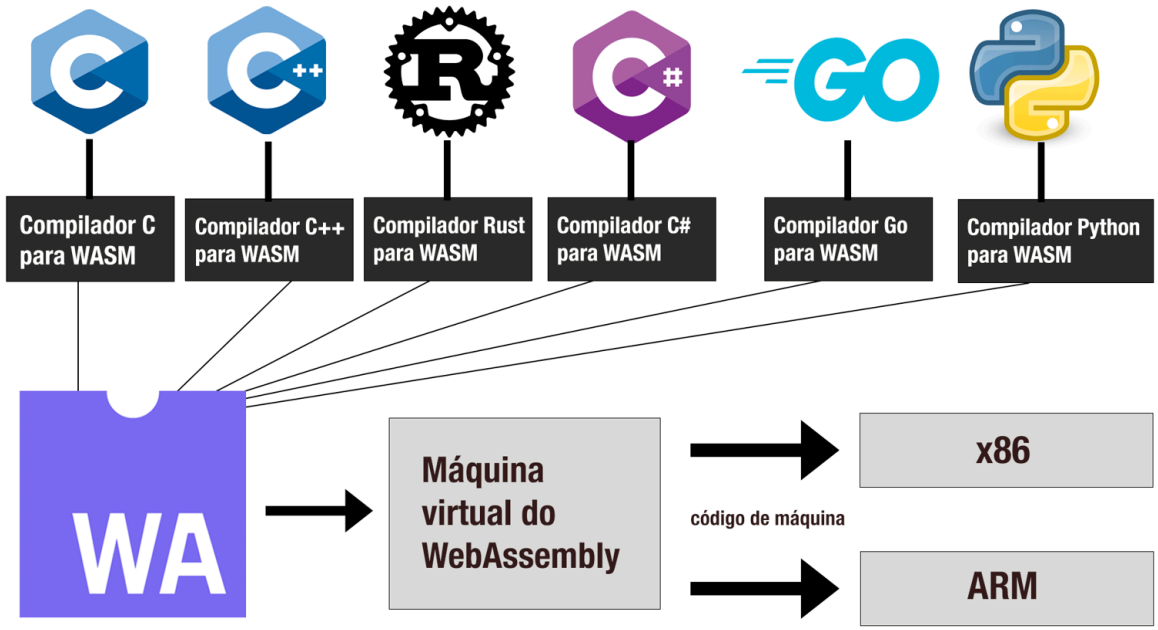


Figura 1.2: Arquitetura do WASM.

Muitos de nós, desenvolvedores(as), gostaríamos de escrever a maior parte do código em uma linguagem de "alto nível" e ainda ter a possibilidade de descer para algo de "baixo nível". A parte mais interessante não é somente descer para algo extremamente otimizado, mas trazer mais diversidade por parte das linguagens ao mundo do desenvolvimento Web. O WebAssembly nos dá um executável alternativo — especificamente projetado para essa finalidade e pronto para ser analisado e executado por qualquer navegador.

Seria então o WebAsssembly o substituto do JavaScript?

É bem comum ouvir este tipo de dúvida, especialmente depois de tudo que vimos até então nas seções anteriores. O WASM tem várias limitações que vamos aprender no decorrer do livro, porém é fundamental destacar desde agora que ele não tem como objetivo substituir o JavaScript, mas, sim, trabalhar em conjunto.

Ambas as tecnologias trabalham em conjunto e resolvem problemas diferentes: o tipo de problema que o WASM resolve são predominantemente tarefas computacionalmente intensivas, como jogos, manipulação de imagens, matemática, física, efeitos de áudio etc. Já o JavaScript você vai usar para outro tipo de problemas na Web, como manipulação do DOM (modificando e construindo visualizações e trabalhando com UIs), uso das APIs da Web etc.

> Um dos maiores limitadores pelo qual o WebAssembly não pode nem vai assumir o cenário do desenvolvimento Web e substituir o JavaScript é o acesso ao DOM (*Document Object Model*, em inglês). Atualmente, o WASM não consegue acessar o DOM e você vai acabar precisando do JavaScript para fazer o acesso.
>
> São muitos anos de melhoria da plataforma em favor de apenas uma linguagem e muitas funcionalidades que foram implementadas. O navegador e JavaScript estão trabalhando juntos por muito tempo e, nesse período, diversas APIs dos navegadores foram criadas. WebAssembly possui limitações quanto a esse tipo de trabalho, o que o JavaScript faz impecavelmente e cada vez melhor, dada a evolução por meio dos padrões definidos pela *Ecma International* ao longo dos anos.

Historicamente, as máquinas virtuais (*virtual machine*, em inglês) dos navegadores conseguiam lidar apenas com JavaScript. Isso funcionou bem durante muitos anos, pois o JavaScript é suficiente para resolver a maioria dos problemas que as pessoas têm na Web

hoje. No entanto, conforme a Web foi expandindo, ela acabou encontrando problemas em alcançar desempenho nativo. Problemas como custo de download, análise e compilação podem sofrer diversas limitações e acabar também resultando em vários gargalos de desempenho.

Além disso, o WebAssembly pode ser facilmente empacotado e distribuído pela sua incrível portabilidade. Existe um gerenciador de pacotes WebAssembly (WAPM, *WebAssembly Package Manager*, em inglês) que permite publicar e distribuir pacotes WebAssembly para serem executados de forma independente em tempo de execução do WASM. Outros gerenciadores de pacotes atuais, como o NPM, também suportam o WebAssembly para fazer parte de seus pacotes JavaScript.

Então, a resposta mais curta e simples em relação à pergunta popular sobre se o WASM vai substituir o JavaScript é: o WebAssembly é uma dentre as muitas ferramentas no cinto de utilidades do Batman (ou, no caso, do(a) desenvolvedor(a) Web). Por exemplo, as ferramentas de quem desenvolve para Web são: HTML (declara a UI), CSS (estiliza a UI), JavaScript (adiciona funcionalidades a uma UI) e, agora, WebAssembly (processa tarefas pesadas e retorna os resultados ao JavaScript que pode entregar a UI).

É notável que ambas as tecnologias se complementam. Você não pode ter WebAssembly funcionando no navegador sem uma pequena camada de JavaScript, pois a Web API envelopa o código executável do WebAssembly exportado com funções JavaScript que podem ser chamadas normalmente, e o código WebAssembly pode importar e chamar funções JavaScript de forma síncrona.

Rust e WASM

Neste livro, vamos evitar a troca de linguagens nos exemplos. Usaremos Rust para compilação de WASM de grande maioria dos exemplos e, posteriormente, o formato de texto do WebAssembly;

assim, mantemos a didática exclusivamente focada no WebAssembly. Mas por que Rust quando temos tantas outras linguagens disponíveis para gerar WebAssembly?

Rust é uma linguagem de programação de sistemas patrocinada pela Mozilla Research, que a descreve como uma "*linguagem segura, concorrente e prática*", que suporta paradigmas funcionais e procedimentos imperativos. Rust é sintaticamente semelhante ao C++, porém foi projetada para fornecer melhor uso de memória enquanto mantém o desempenho de alta performance.

A linguagem é livre da coleta de lixo não determinística e dá a programadores o controle sobre indireção, monomorfização e *layout* da memória. Por esse motivo, afeta positivamente o tamanho do código, cujo grau de importância é gigante, dado que o `.wasm` deve ser baixado pela rede.

Rust quase não possui tempo de execução, o que permite que `.wasm` tenha um tamanho pequeno, pois não há inchaço extra incluído no código como um coletor de lixo. Você só paga (em tamanho de código) pelas funções que realmente usa.

> Caso você queira saber mais sobre Rust, confira o livro: *Rust: Concorrência e alta performance com segurança*, de Marcelo Castellani e Willian Molinari (https://www.casadocodigo.com.br/products/livro-rust).

Inúmeras ferramentas já existentes do ecossistema de desenvolvimento JavaScript estão integradas com Rust e WebAssembly. Isso significa que você pode continuar usando ferramentas populares que você já usa no desenvolvimento com JS, como NPM, Turbopack, Parcel, Webpack e outras. Muitos empacotadores populares do JavaScript permitem até mesmo compilar Rust para WebAssembly.

Optamos por usar recursos simples da linguagem Rust nos exemplos, pois o foco deste material é a aprendizagem dos fundamentos do WebAssembly e os conceitos computacionais que existem ao redor do WebAssembly. Isso foi pensado para que você pudesse ter a liberdade de escolha sobre qualquer linguagem após a leitura do livro, e não ficar preso(a) somente a ferramentas existentes no universo do Rust.

Um detalhe importante é que, no decorrer do livro, assim que os conceitos do WASM estiverem enraizados, eventualmente substituiremos Rust e passaremos a escrever aplicações WebAssembly usando seu respectivo formato de texto. Entretanto, mesmo que posteriormente a compilação para WASM seja realizada com sua representação de texto, é correto afirmar que usaremos Rust na grande maioria dos exemplos do livro.

Dito isto, podemos iniciar nossa jornada de aprendizado. No próximo capítulo, vamos entender melhor sobre o WebAssembly e sua comunicação com o JavaScript. Vamos destrinchar mais sobre o formato binário e os fundamentos para escrevermos nossa primeira aplicação usando WebAssembly.

Capítulo 2
Fundamentos ao primeiro módulo

Antes de começarmos este capítulo, quero destacar que, de agora em diante, até o final do livro, lidaremos com exemplos práticos, e se você tiver qualquer curiosidade ou precisar verificar algo, todos os exemplos do livro (incluindo os arquivos `.wasm`) estão disponíveis no GitHub, pelo link https://github.com/raphamorim/livro-webassembly-exemplos. Basta clonar o repositório usando git ou fazer uso da opção de download do código-fonte que o GitHub provê.

Neste capítulo, vamos aprender os fundamentos básicos do WebAssembly, analisando de forma contemplativa o processo de criação do formato binário até sua execução no navegador. Por fim, realizaremos a escrita e execução do nosso primeiro módulo WebAssembly, que será usado como referência no próximo capítulo para explicar de maneira detalhada como o JavaScript trabalha com o WASM.

Na introdução vimos sobre a história e a definição do que é o WebAssembly, e o entendimento de ambos torna a visualização do objetivo da tecnologia mais clara, porém ainda há uma lacuna a ser preenchida a respeito do WASM: como funciona a leitura do código binário em um navegador criado para lidar exclusivamente com JavaScript?

Essencialmente, o WASM é apenas uma máquina virtual que roda em todos os navegadores modernos. Por esta razão, você pode escrever aplicações para a Web com a representação textual do WebAssembly ou em qualquer linguagem de programação que tenha permitido a compilação para o formato de arquivo binário.

2.1 Máquina virtual do WebAssembly

Antigamente, todo mundo implantava código diretamente em computadores físicos, porém, conforme as demandas passaram a exceder a capacidade da infraestrutura que existia na época, tivemos a evolução para máquinas virtuais. O uso de máquinas virtuais permitiu rodar inúmeras máquinas em um único hardware, e isso deu origem aos *Data centers*. Eventualmente excedemos a capacidade dos *Data centers* para lidar com a implantação de inúmeras máquinas virtuais, então surgiu a virtualização com *Docker*. Atualmente, é comum encontrar infraestruturas baseadas em imagens do Docker que são executadas por cima das máquinas virtuais implantadas em computadores físicos.

O WebAssembly não é considerado apenas uma solução para viabilizar código de outras linguagens de programação para o navegador. Também é considerado por muitos como parte da próxima evolução da infraestrutura na nuvem pelas características de sua máquina virtual. Inúmeras empresas já adotaram WASM como parte de suas soluções de infraestrutura, como a própria Microsoft.

Historicamente, evoluímos de discos rígidos com imagens personalizadas para pacotes inteligentes em máquinas físicas para máquinas virtuais e, posteriormente, para imagens do *Docker*; e, agora, temos uma unidade de implantação menor, mais rápida, mais portátil e mais segura: o módulo WebAssembly.

A máquina virtual do WebAssembly é baseada em pilha e toda máquina (virtual ou não) tem um conjunto básico de instruções primitivas que podem ser usadas para dizer o que fazer. No decorrer do livro, vamos ver como ela funciona e todas as suas respectivas instruções. Toda linguagem de programação de alto nível, em última análise, tem como a saída do seu respectivo compilador um arquivo com código de nível de máquina que pode ser executado em diferentes plataformas.

O conjunto de instruções usado pelo WebAssembly é portátil. Isso significa que é impossível expressar algo no código de bytes do WebAssembly que funcionará em uma plataforma, mas não em outra. O WebAssembly possui seu próprio conjunto de instruções, mas a máquina responsável por executá-las é virtual, ou seja, é apenas mais um processo em seu computador. Isso tem um impacto profundo e frequentemente subestimado na importância do WebAssembly.

Máquinas virtuais baseadas em pilha, como a do WebAssembly, são rápidas porque o código para gerenciar pilhas e executar operações nelas é realmente muito simples e pode ser altamente otimizado. Os arquivos binários do WebAssembly também são pequenos. Dependendo do que você está construindo, os arquivos binários são medidos em *kilobytes*, não em *megabytes* ou *gigabytes*.

A leitura e processamento do código binário por parte das máquinas virtuais WASM dependem dos fundamentos básicos que fazem parte da especificação do WebAssembly, sem os quais é inviável criar qualquer tipo de programa executável.

2.2 Fundamentos básicos

O WebAssembly possui diversos conceitos em sua especificação e, dentre eles, alguns são considerados conceitos-chaves. São fundamentos básicos ao redor dos quais os demais tópicos orbitam. No decorrer do livro, vamos aprender todos os conceitos por trás do WASM, mas, por agora, começaremos definindo os fundamentos básicos.

Módulo

Os programas WebAssembly são organizados em módulos, que são a unidade de implantação, carregamento e compilação. Um módulo

possui as definições de tipos, funções, tabelas, memórias, globais, importações e exportações. O módulo também tem poder para fornecer a inicialização de dados e elementos ou até mesmo uma função inicial.

Fundamentalmente, o módulo representa um código binário do WebAssembly que foi compilado, tornando-se um executável. Módulos não possuem nenhum estado e, por esse motivo, assim como um *Blob* (objeto provido pelo JavaScript semelhante a um arquivo de dados brutos imutáveis), pode ser lido como texto ou dados binários.

Memória

A memória é representada como uma matriz contígua e mutável de *bytes* não interpretados e pode crescer dinamicamente. Uma memória criada pelo JavaScript ou pelo código WebAssembly é acessível e mutável por ambos. Os *bytes* da memória podem ser alterados pelo módulo, por meio de instruções de memória ou no tempo de execução do programa que estiver executando WASM (depende da configuração do módulo). A API do WebAssembly no JavaScript representa seu tipo por um *ArrayBuffer*.

Os índices de uma matriz de memória linear podem ser considerados endereços de memória, e as instruções que precisam acessar um local de memória recebem um deslocamento relativo ao início da memória. Esse comportamento garante características importantes na segurança da memória no WebAssembly (supondo que a implementação do tempo de execução esteja livre de erros).

Uma vez que o tamanho de uma memória é sempre definido e conhecido em tempo de compilação, é possível verificar em tempo de execução se um deslocamento de memória que um módulo está

tentando acessar ainda está dentro dos limites de sua memória alocada.

Outra característica é que um módulo não pode acessar a memória de outro módulo, a memória do tempo de execução ou a memória do sistema operacional subjacente, a menos que tenha acesso explicitamente. Isso reflete também em restringir módulos potencialmente maliciosos de usarem endereços de memória arbitrários para acessar dados fora de sua configuração de memória linear.

Tabela

É uma matriz de referências tipadas redimensionáveis que não podem ser armazenadas como *bytes* brutos na memória (por motivos de segurança e portabilidade). As referências da tabela podem ser funções externas ou internas com diferentes tipos de assinatura. Assim como a memória, a tabela pode ser criada pelo JavaScript e pelo WebAssembly, além de poder ser também acessada e mutável por ambos.

Tabelas são extremamente úteis em cenários onde o WebAssembly lida com funções que podem chamar outras funções dinamicamente em tempo de execução, especialmente em casos que a especificação da função a ser executada depende de qualquer tipo de valor.

Global

É um objeto que representa uma instância de variável global, acessível pelo JavaScript e por uma ou mais instâncias de um módulo. Cada global armazena um único valor do tipo global fornecido. A variável global pode especificar ou não se um global é imutável ou mutável. Os globais são referenciados por meio de índices globais, começando sempre com o menor índice disponível possível.

Instância

É uma instância executável do módulo. O processo de instanciamento verifica se o módulo é válido e se as importações fornecidas correspondem aos tipos declarados e, caso contrário, pode falhar com um erro.

O instanciamento também pode resultar em um erro personalizado do WASM, conhecido como *armadilha*, quando tentar inicializar uma tabela ou memória de um segmento ativo ou na execução da função de início. Cabe ao incorporador, responsável pela configuração e instalação do módulo, definir como essas condições serão reportadas.

Na API do JavaScript para lidar com WASM, o processo de instanciamento pode receber opcionalmente como argumento um objeto com estado. O estado do objeto pode incluir uma propriedade de memória, uma tabela e um conjunto de valores importados.

Assim que a instância é criada, assumindo que o módulo instanciado seja executável sem erros, o objeto que representa a instância será similar a um módulo JavaScript (*ES2015*) que foi carregado em um determinado escopo. O valor da instância possui todas as funções exportadas do WebAssembly.

Instâncias, módulos, tabelas e globais são fundamentos que são utilizados frequentemente por compiladores de outras linguagens de programação que visam WASM como destino de compilação ou na representação textual do WebAssembly. Esses fundamentos estão também presentes na API que o JavaScript provê tanto no navegador quanto no lado do servidor, por exemplo, no NodeJS.

Veja a seguir um exemplo do `WebAssembly.Instance` no console do navegador:

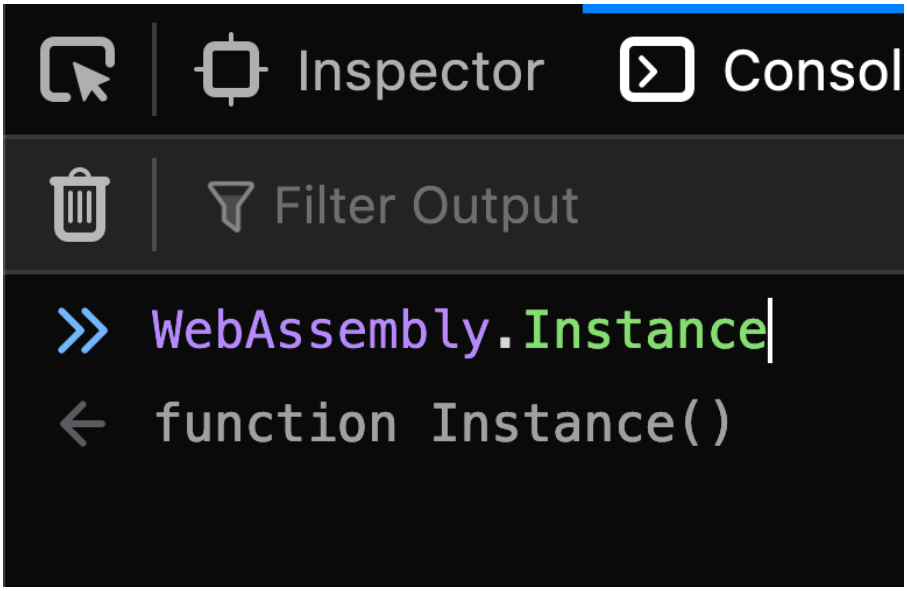


Figura 2.1: Exemplo da classe Instance no navegador.

O processo de instanciamento é dependente de um módulo WebAssembly ser válido, caso contrário a criação da instância vai falhar na primeira etapa de validação. Os módulos são criados a partir do formato de arquivo binário e precisam ser condizentes com as regras da especificação, além de fazerem uso coerente dos tipos providos pelo WASM.

2.3 Formatos de arquivo

As extensões de arquivo do WebAssembly são `.wasm` e `.wat`. Historicamente, `.wat` era `.wast`, mas agora o formato é somente usado pelo conjunto de testes da especificação. O formato de arquivo binário `.wasm` não foi projetado para ser lido por seres humanos. Entretanto, o formato `.wat` é uma representação textual legível por humanos, que se assemelha a um cruzamento entre uma lista aninhada de dados e linguagens *Assembly* tradicionais.

A fim de entender melhor a diferença entre esses formatos, vamos visualizar o seguinte código em Rust, uma simples função de soma

de inteiros que retorna um número inteiro com sinal de 32 bits:

```
pub fn soma(inteiro_a: i32, inteiro_b: i32) -> i32 {
  inteiro_a + inteiro_b
}
```

O formato de texto `.wat` gerado deve ser semelhante ao código a seguir, porém nem sempre é o mesmo, já que varia de acordo com o comportamento do compilador utilizado e dos parâmetros de compilação configurados:

```
(module
 (table 0 anyfunc)
 (memory $0 1)
 (export "memory" (memory $0))
 (export "_Z4somaii" (func $_Z4somaii))
 (func $_Z4somaii (param $0 i32) (param $1 i32) (result i32)
  (i32.add
   (get_local $1)
   (get_local $0)
  )
 )
)
```

Em outra mão, temos o (assustador) código binário `.wasm` expresso a seguir usando Firefox x86 Assembly e JIT como linha de base:

```
wasm-function[0]:
  sub rsp, 0x18                       ; 0x000000 48 83 ec 18
  mov qword ptr [rsp + 8], r14        ; 0x000004 4c 89 74 24 08
  mov rax, rsp                        ; 0x000009 48 8b c4
  add rax, 0                          ; 0x00000c 48 05 00 00 00 00
  cmp qword ptr [r14 + 0x28], rax     ; 0x000012 49 39 46 28
  jae 0x34                            ; 0x000016 0f 83 18 00 00 00
 0x00001c:
  mov dword ptr [rsp + 4], edi        ; 0x00001c 89 7c 24 04
  mov dword ptr [rsp], esi            ; 0x000020 89 34 24
  mov eax, dword ptr [rsp + 4]        ; 0x000023 8b 44 24 04
  mov ecx, dword ptr [rsp]            ; 0x000027 8b 0c 24
  add ecx, eax                        ; 0x00002a 03 c8
```

```
 mov eax, ecx                          ; 0x00002c 8b c1
 jmp 0x3d                              ; 0x00002e e9 0a 00 00 00
0x000033:
 int3                                  ; 0x000033 cc
0x000034:                              ; 0x000034 from: [0x000016]
 add rsp, 0x10                         ; 0x000034 48 83 c4 10
 jmp 0x6e                              ; 0x000038 e9 31 00 00 00
0x00003d:
 mov r14, qword ptr [rsp + 8]          ; 0x00003d 4c 8b 74 24 08
 nop                                   ; 0x000042 66 90
 add rsp, 0x18                         ; 0x000044 48 83 c4 18
 ret
```

Esse arquivo faz uso do *Firefox 86x*, que permite uma saída "desmontada" do binário pelo motor do WebAssembly para ajudar na leitura do formato binário. A leitura desse código seria mais complicada se não fizéssemos uso do *Firefox 86x*, pois teríamos o binário no formato "montado" como a seguir:

```
0061 736d 0100 0000 0187 8080 8000 0160
027f 7f01 7f03 8280 8080 0001 0004 8480
8080 0001 7000 0005 8380 8080 0001 0001
0681 8080 8000 0007 9680 8080 0002 066d
656d 6f72 7902 0009 5f5a 3473 6f6d 6169
6900 000a 8d80 8080 0001 8780 8080 0000
2001 2000 6a0b
```

Uma simples comparação entre os dois formatos de arquivo (`.wat` e `.wasm`) como vimos nesta seção torna a diferença entre ambos os formatos claramente visível. O código binário do WASM foi feito para ser lido e processado da forma mais rápida possível, mas não por você e sim, pela máquina.

Observe também que, no arquivo do formato de texto `soma.wat` , é possível ver o tipo de dados utilizado na função `soma` que foi compilada com o nome personalizado `_Z4somaii` . Assim como no arquivo `soma.rs` , a representação textual do WebAssembly faz a declaração do tipo `i32` , que especifica o uso do número inteiro com sinal de 32 bits.

2.4 Tipos de valores de dados

Os possíveis tipos de valores do WebAssembly são validados no processo de validação, instanciamento e, possivelmente, na execução. Os programas WebAssembly operam em valores numéricos primitivos e são classificados em: *Bytes*, vetores, nomes, números inteiros, números de ponto flutuante.

> Você pode ver a lista completa dos valores do WebAssembly na sua especificação em inglês: https://webassembly.github.io/spec/core/syntax/values.html.

Os valores do tipo *Bytes* são a forma mais simples de qualquer valor existente no WASM e são basicamente *bytes* brutos não interpretados. Na sintaxe abstrata da representação textual (formato de arquivo `.wat`), eles são representados com hexadecimais.

Os *vetores* são valores de 128 bits que são processados por instruções vetoriais (SIMD), e *nomes* são basicamente sequências de caracteres definidos com uma codificação de dados personalizada (*Unicode*). Nomes possuem algumas limitações baseadas na especificação do formato binário, mas não vamos entrar em detalhes agora, pois conversaremos sobre esses dois tipos de valores posteriormente.

Temos também os dados numéricos, que podem ser valores de números *inteiros* ou números de *ponto flutuante*. Vimos um exemplo de dado numérico na função `soma` da seção anterior quando utilizamos o valor `i32`.

O WebAssembly suporta basicamente apenas quatro tipos de dados numéricos:

- Números inteiros de 32 bits (`i32`);
- Números inteiros de 64 bits (`i64`);
- Números de ponto flutuante (*floats*) de 32 bits (`f32`);

- Números de ponto flutuante (*floats*) de 64 bits (`f64`).

No núcleo interno do WASM, não há distinção entre tipos inteiros assinados e não assinados — por exemplo, não existem tipos como `u8` , `i16` e outros. Em vez disso, inteiros são interpretados pelas respectivas operações como sem sinal ou com sinal na representação de complemento de dois. Isso significa que compiladores que têm WebAssembly como destino de compilação precisam lidar com a conversão de tipos por conta própria.

Apesar de possuir diversos tipos de valores de dados, apenas números inteiros e números flutuantes podem ser computados em uma variável, e isso basicamente limita o uso de outros tipos. Por esse motivo, é comum ouvir que o WebAssembly apenas suporta quatro tipos de valores (`i32` , `i64` , `f32` e `f64`).

Em virtude de termos apenas quatro tipos de valores que podem ser computados sem limitações na máquina virtual do WASM, alguns conceitos da Ciência da Computação (por exemplo, codificação de dados) se tornam necessários para qualquer aplicação.

Isso significa que diferentes tipos dados precisam ter o tratamento customizado. Por exemplo, uma simples função escrita em Rust que retorna um tipo *String* quando compilada para WebAssembly teria que possuir o tratamento da codificação de valores corretamente se desejar ler ou manipular no lado do WebAssembly.

```
fn minha_linda_funcao() -> String {
  String::from("O valor desta String precisa ser codificado...")
}
```

A boa notícia é que a maioria das linguagens de programação provê ferramentas que auxiliam com a codificação de dados. Futuramente, vamos ver como usar dados numéricos para trabalhar com imagens, *strings*, estruturas de dados customizadas, entre outros tipos. Todavia, não podemos pular para conversar sobre codificação e decodificação de dados sem entender alguns tópicos, por exemplo, como o processo de compilação para WASM funciona.

2.5 Compilação para o código binário

O ecossistema de desenvolvimento com WebAssembly ainda está em um estágio inicial, e é certo que mais ferramentas surgirão daqui para a frente. No momento, existem diversas opções para criação e desenvolvimento de aplicações com WASM. Algumas opções são:

- Escrever uma aplicação em uma linguagem que possui um compilador que permite a saída de compilação como WebAssembly. Existem diversas linguagens que possuem esse suporte: Rust, Golang, C#, Lua, entre outras;
- Usar *AssemblyScript*, que é semelhante ao *TypeScript* e compila para o binário do WebAssembly;
- Portar um aplicativo C ou C++ com *Emscripten* (compilador baseado na LLVM e Clang que compila código-fonte em C e C++ para WebAssembly);
- Escrever WebAssembly usando sua representação textual, o formato de arquivo `.wat`.

Neste livro, seguiremos duas das opções listadas: vamos escrever código em Rust com destino de saída de compilação configurado para WASM e, posteriormente, vamos trocar Rust pelo formato de texto do WebAssembly. Dito isso, vamos ver como a compilação para WASM funciona com Rust.

Existem alguns destinos de compilação que você pode especificar para o compilador Rust (conhecido como `rustc`) compilar para WASM. Alguns dos destinos de compilação para WebAssembly são:

- `wasm32-unknown-unknown` — Destino de compilação nível 2: esse destino está focado na produção de arquivos binários `*.wasm`. No entanto, a biblioteca padrão do Rust é ignorada. Em razão de a definição do destino conter a especificação "*unknown*" (desconhecido, em inglês), isso reforça que o *libstd* (biblioteca padrão) não pode assumir nada. Embora os binários provavelmente funcionem por tempo indeterminado, funcionalidades comuns como `println!` ou `panic!` não vão

funcionar. Esse destino de compilação foi introduzido oficialmente no Rust em dezembro de 2017.

- `wasm32-wasi` — Destino de compilação nível 2: a maior diferença entre `wasm32-unknown-unknown` e `wasm32-wasi` é a portabilidade das API do sistema. O *WASI* basicamente especifica como APIs de sistemas podem se comunicar com o WebAssembly, disponibilizando, por exemplo, operações de IO. É importante destacar que navegadores ainda não suportam WASI. Veremos sobre WASI mais adiante.

- `wasm32-unknown-emscripten` — Destino de compilação nível 2: compila para WebAssembly usando o *Emscripten*. É uma opção pouco popular atualmente, dada a existência do `wasm32-unknown-unknown` e `wasm32-wasi`. Entretanto, pela sua arquitetura, ainda possui valor para compilação de programas já existentes (tipo de problema que o Emscripten é focado em resolver) para a Web, especialmente se houver uso do SDL2 (*Simple DirectMedia Layer 2*) e OpenGL.

- `wasm64-unknown-unknown` — Destino de compilação nível 3: lida com a biblioteca padrão do Rust da mesma forma que `wasm32-unknown-unknown`, porém compila com endereçamento de memória 64 bits em vez de 32. Ainda não é um destino de compilação estável e está em desenvolvimento. Se você quiser, pode ver a especificação neste link: https://github.com/webassembly/memory64.

- `wasm64-wasi` - Destino de compilação experimental. De forma similar aos destinos `unknown-unknown`, opera a mesma funcionalidade do WASI, porém com endereçamento de memória 64 bits.

Ao decorrer dos anos, o destino de compilação `wasm32-unknown-unknown` se tornou a opção mais popular para compilação focada em WebAssembly e isso se manteve até os dias atuais. Neste livro, usaremos primariamente os destinos `wasm32-unknown-unknown` e `wasm32-`

`wasi` . Dito isso, que tal começarmos instalando Rust e os destinos de compilação?

Se você está usando MacOS, Linux ou outro sistema operacional do tipo Unix, basta abrir o terminal e executar este código para iniciar a instalação (se necessário, há mais informações no site oficial, https://www.rust-lang.org/tools/install):

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Esse código vai instalar uma ferramenta chamada `rustup` , que permite gerenciar várias versões do Rust, e, por padrão, ele instala a versão estável mais recente. O `rustup` vai instalar tanto o compilador oficial do Rust, o `rustc` , quanto o `cargo` , o gerenciador de pacotes do Rust. Neste livro, usamos a versão 1.70.0 do compilador.

Para instalar um destino de compilação, usamos o `rustup` onde especificamos que queremos adicionar um destino juntamente com seu respectivo nome. O código a seguir exemplifica a instalação do destino de compilação `wasm32-unknown-unknown` :

```
rustup target add wasm32-unknown-unknown
```

Se desejar, podemos fazer o mesmo processo com o destino de compilação `wasm32-wasi` , apesar de que não vamos usar este destino tão cedo no livro:

```
rustup target add wasm32-wasi
```

Pronto, você está com o ambiente configurado e preparado para compilar para WebAssembly usando Rust.

2.6 Escrevendo nosso primeiro módulo

Imagino que você esteja querendo colocar em prática as ferramentas que acabamos de instalar na seção anterior. Vamos criar juntos nosso primeiro módulo WebAssembly usando Rust!

Vamos colocar a mão na massa. Já que veremos nos próximos capítulos como funciona a API do JavaScript para lidar com WebAssembly, faremos uso de duas ferramentas especiais que facilitam e agilizam o desenvolvimento com WASM: `wasm-pack` e `wasm-bindgen`.

O `wasm-pack` é uma opção bastante popular para trabalhar com WASM, por diferentes motivos: é parte do grupo de ferramentas oficiais que são mantidas pelos próprios mantenedores da linguagem; o uso do empacotador é recomendado pela própria Mozilla e foca exclusivamente no desenvolvimento com JavaScript.

O `wasm-bindgen` é um pacote que atua como um "canivete suíço" do WebAssembly e provê inúmeras — com ênfase em quantidade — opções de operações quando é colocado em uso, entre elas: a codificação de dados, a leitura de valores do *DOM*, a execução de funções JavaScript no Rust (por exemplo, `window.alert`), inicialização da instância e tantas outras operações. Pode ser usado separadamente ou em conjunto com o `wasm-pack`. Esse pacote facilita muito as interações de alto nível entre módulos WebAssembly e JavaScript.

Usaremos essas duas ferramentas apenas neste capítulo, então é bom não se acostumar à praticidade que ambas ferramentas proveem. O livro tem como objetivo mostrar como o WebAssembly funciona de fato e isso significa ver literalmente todas as partes por baixo dos panos.

O primeiro passo é instalar o `wasm-pack` usando o *Cargo*:

```
cargo install wasm-pack
```

Vamos criar um novo projeto Rust para o nosso módulo WebAssembly, e para isso também precisamos do *Cargo*. O comando

a seguir, quando executado, criará um diretório chamado `modulo`:

```
cargo new modulo --lib
```

Dentro do diretório `modulo` existe um arquivo chamado `Cargo.toml`, que é responsável por definir certas configurações e preferências do projeto, assim como a lista de dependências. Vamos editar o `Cargo.toml` para adicionar mais uma dependência: o `wasm-bindgen`. O arquivo deverá ficar assim:

```
[package]
name = "modulo"
version = "0.1.0"
edition = "2021"

[lib]
crate-type = ["cdylib"]

[dependencies]
wasm-bindgen = "0.2.84"
```

A configuração `crate-type` usando `cdylib` significa que você quer que o compilador (`rustc`) crie uma biblioteca de sistema dinâmica. Isso é geralmente usado para construir bibliotecas que podem ser vinculadas a programas C e C++.

E por que a configuração de saída não é `.dll`, `.so` ou `.dylib`? Isso se deve ao fato de que não estamos compilando para Windows, Linux ou MacOS. A compilação é para `wasm32-unknown-unknown` e é importantíssimo configurarmos como uma biblioteca dinâmica, pois bibliotecas dinâmicas (`cdylib`) podem ser carregadas em tempo de execução, por exemplo, pelo navegador.

Essencialmente, para o nosso primeiro módulo, queremos simplesmente criar um arquivo `.wasm` sem uma função de início e que permita exportar funções de forma dinâmica.

Como o WebAssembly suporta apenas os tipos `i32`, `f32`, `i64` e `f64`, se você quiser trabalhar com qualquer outro tipo, como uma

`String` ou um `Map`, é preciso primeiro codificar os dados do tipo usado e provavelmente especificar uma decodificação onde você for ler esses dados (conversamos brevemente sobre isso na seção *Tipos de valores de dados*).

No entanto, `wasm-bindgen` cuida dessa operação para você automaticamente e não há necessidade de se preocupar com isso. Assim que o `wasm-bindgen` verifica a notação da função que retorna uma *String*, cuida de criar uma lógica de codificação e decodificação tanto no lado do Rust (no momento da compilação) quanto no JavaScript (no tempo de execução).

Com o `wasm-bindgen` instalado, dentro no arquivo `lib.rs` na pasta `src`, criaremos uma simples função nomeada como "texto" que retorna uma `String`:

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn texto() -> String {
  String::from("Bem-vindo ao fantástico mundo do WebAssembly.")
}
```

Poderíamos compilar esse código usando diretamente o compilador `rustc` ou até mesmo com o `cargo`. Porém, já que estamos usando o `wasm-pack` neste módulo, vamos usar seu comando de construção e empacotamento. Execute o comando a seguir no seu respectivo terminal na raiz da pasta `modulo`:

```
wasm-pack build --target web
```

Após a execução do comando, um diretório `pkg` será criado com nossa biblioteca JavaScript contendo a função que escrevemos no Rust compilada no formato binário, como uma função JavaScript. O mesmo comando também gera os tipos do destino de compilação em *TypeScript*.

Se você visualizar o arquivo `pkg/olamundo.js` (que vamos importar daqui a pouco), perceberá que já possui o tratamento correto do

WebAssembly usando a API do navegador. O módulo e a instância já foram automaticamente criados para você sem nenhum esforço!

Outro ponto importante para destacar é que estamos usando explicitamente o parâmetro de destino (`--target`) configurado como `web` . Entretanto, existem diferentes parâmetros que podemos usar com `wasm-pack` , mas depende de como queremos usar o arquivo binário `.wasm` eventualmente. Veja a seguir a lista de outras opções possíveis:

1. `--target bundler` : destino configurado para empacotadores (*bundlers*, em inglês) como Webpack, Parcel ou Rollup;
2. `--target web` : destino configurado para a Web como um módulo ES2015 JavaScript;
3. `--target no-modules` : destino configurado para a Web, porém não sendo um módulo ES2015 JavaScript;
4. `--target nodejs` : destino configurado para o NodeJS.

Para usar o arquivo binário que acabamos de compilar no JavaScript, podemos importar o módulo `pkg` gerado para nosso projeto que contém uma função chamada `init` , que faz a validação e a respectiva instância do módulo WebAssembly. Esse módulo JavaScript também é uma cortesia do `wasm-pack` e `wasm-bindgen` .

Vamos então colocar em teste prático nosso pacote! Para isso, criaremos um `index.html` na raiz do projeto:

```
<!DOCTYPE html>
<html>
  <head>
  <meta charset="UTF-8" />
  <title>Primeiro Módulo</title>
  <script type="module">
    /*
    * O pacote por padrão exporta a função "init" que
    * inicializa o carregamento do WASM.
    * Também exporta a função que escrevemos em
    * Rust nomeada "texto"
```

```
    */
    import init, { texto } from './pkg/modulo.js';

    // Função assíncrona de execução
    async function executar() {
      // Inicializa o carregamento
      await init();

      // Coloca o resultado no corpo da página
      document.body.textContent = texto();
    }

    executar();
  </script>
  </head>
  <body></body>
</html>
```

Perfeito! Estamos quase lá. Agora precisamos de um servidor HTTP que possa servir nossos arquivos estáticos, pois o binário `.wasm` é baixado e processado no arquivo `modulo.js` carregado no `index.html`.

Para criar um simples servidor, existem diferentes opções:

- Já que instalamos o cargo anteriormente, você pode simplesmente executar na raiz do diretório: `cargo install cargo-server && cargo server --port 5000`;
- Se você possuir NodeJS e NPM instalados, pode executar `npm install -g npx && npx serve .` (com o ponto final no comando) na raiz do diretório;
- Se você possuir Python 3 instalado (como a maioria das distribuições Linux e as versões mais atuais do MacOs), basta executar `python -m http.server` na raiz do diretório;
- Se você possuir Python 2.7 instalado, basta executar `python -m SimpleHTTPServer` na raiz do diretório.

Após inicializar o servidor localmente, acesse `http://localhost:5000` na aba do seu navegador (verifique se a porta usada na criação do

servidor é `5000`; do contrário, é necessário usar a porta configurada). O resultado deverá ser igual à imagem a seguir:

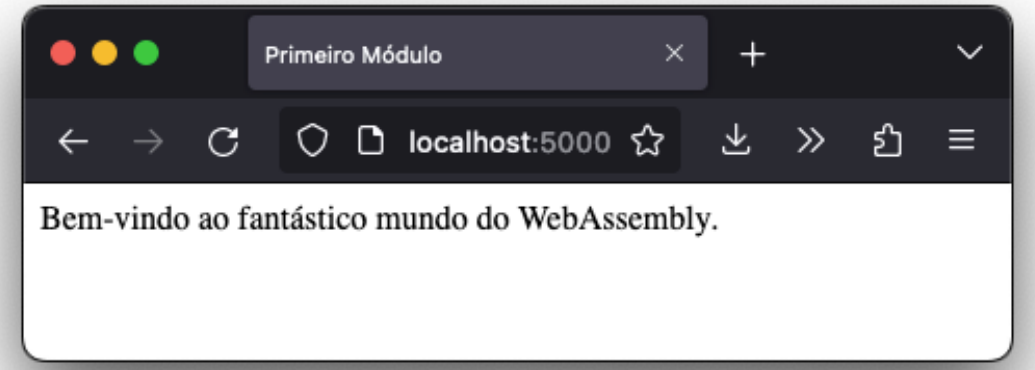


Figura 2.2: Resultado do primeiro módulo no navegador.

Parabéns, você acabou de escrever seu primeiro módulo WebAssembly! No próximo capítulo, entenderemos melhor o processo por parte do navegador, especialmente focado nos arquivos dentro da pasta `pkg` que foi criada automaticamente.

Daqui para a frente não usaremos mais `wasm-pack` e `wasm-bindgen`, então sinta-se livre para desinstalar esses pacotes, se desejar. Até o final do livro compilaremos para WASM fazendo uso do `cargo`, `rustc` ou do pacote oficial de utilitários binários do WebAssembly.

Capítulo 3
Por trás da magia no navegador

No último exemplo do capítulo anterior, tivemos a ajuda de algumas ferramentas (`wasm-pack` e `wasm-bindgen`) para facilitar e agilizar o desenvolvimento com WebAssembly. Elas auxiliam na escrita, de forma automática, de blocos de códigos que são responsáveis por carregar e instanciar módulos para cada arquivo binário usado na aplicação. Entretanto, o objetivo deste livro é tornar claro o entendimento de todos os conceitos que envolvem o WebAssembly, e isso inclui também a API que o JavaScript provê. Dito isso, aprenderemos toda a "magia" que estas ferramentas providenciaram anteriormente — por exemplo, como ocorre o instanciamento de um módulo, a definição de uma ou mais tabelas, configuração da memória, globais, entre outros.

Essencialmente, a forma como o WebAssembly funciona no navegador é baseada na leitura e escrita de dados brutos feita pelo JavaScript. Os dados brutos são, basicamente, qualquer tipo de dado não processado, podendo ser até mesmo arquivos binários. O JavaScript provê um recurso para trabalhar com dados brutos no navegador e até mesmo em servidores usando NodeJS, conhecido como matrizes tipadas (*Typed Arrays*).

3.1 Matrizes tipadas

Originalmente, o JavaScript foi criado com objetos do tipo *Array*. Esses objetos mudam de tamanho dinamicamente e podem ter qualquer valor. O motor do JavaScript também aplica otimizações para que *Arrays* sejam rápidos e, por isso, são amplamente usados no desenvolvimento Web.

No entanto, o objeto do tipo *Array* não foi criado pensando em lidar com recursos que apareceram com o passar do tempo na internet, como manipulação de áudio e vídeo, acesso a arquivos binários usando WebGL, representação de estruturas em memória etc. O JavaScript acabou precisando evoluir mais uma vez para poder manipular dados binários brutos de forma rápida e prática. Por esse motivo, foi criada a matriz tipada.

Matrizes tipadas são objetos do tipo *Array* que fornecem um mecanismo para leitura e gravação de dados binários brutos em *buffers* de memória. É um conceito que é basicamente o "coração" do processamento de arquivo binário, além de ser usado em diversas outras APIs do navegador, incluindo o WebAssembly.

As matrizes tipadas são utilizadas por diversas APIs do navegador, como WebGL, Canvas, API de áudio, *fetch*, XMLHttpRequests, WebSockets, Web Workers, API de fonte de mídia, APIs de arquivo e outras. A maioria dos casos de uso das matrizes tipadas são justamente para lidar com o trabalho multimídia sensível ao desempenho, bem como para a transmissão de dados de maneira eficiente.

> A matriz tipada é um recurso da linguagem JavaScript que não é popular entre desenvolvedores Web, mas é usado amplamente pela API do WebAssembly.

Uma vez que a matriz tipada é criada em cima de uma memória bruta, o motor do JavaScript pode passar a memória diretamente para bibliotecas nativas sem ter que converter meticulosamente os dados em uma representação nativa. Por esse motivo, matrizes tipadas são bem mais rápidas que um *Array* para passar dados para WebGL e outras APIs que lidam com dados binários.

É importante destacar que as matrizes tipadas não são *Arrays*. O valor de qualquer matriz tipada verificada com o uso do método `Array.isArray()` será falso. Além disso, os métodos que um *Array*

tradicional possui, como `push` e `pop`, são inexistentes em uma matriz tipada. As matrizes tipadas são constituídas por *buffers* de memória bruta e sua respectiva visualização.

Buffers e visualizações

Um ***buffer*** (implementado pelo objeto `ArrayBuffer` no JavaScript) é um objeto que representa um bloco de dados brutos. Basicamente, ele aloca um espaço específico em memória, mas não oferece nenhum mecanismo para acessar seu conteúdo. Para acessar os dados contidos em um *buffer*, você precisa criar uma visualização.

A visualização fornece um contexto dos dados em memória que é responsável por dizer qual o tipo de dados, o deslocamento inicial e o número de elementos que a memória possui. Após a associação da visualização ao bloco de dados brutos (`ArrayBuffer`), nasce uma matriz tipada.

Que tal vermos um exemplo prático para poder facilitar a visualização do que conversamos até agora sobre *buffers* e visualizações? No código a seguir, criamos algumas matrizes tipadas a partir do mesmo *buffer* de memória. Existe mais de uma forma de criar um *ArrayBuffer*, mas aqui vamos fazê-lo da forma mais explícita possível: primeiro criamos o *ArrayBuffer* e depois adicionamos visualizações:

```
// Array Buffer de 256 bytes
const bufferMemoria = new ArrayBuffer(256);

// Exemplo de visualização inteira
const visualizacaoInteira = new Uint8Array(bufferMemoria);

// Outros exemplos de visualizações que especificam o
// deslocamento inicial e o número de elementos
const visPrimeiraMetade = new Uint8Array(bufferMemoria, 0, 128);
const visTerceiroQuarto = new Uint8Array(bufferMemoria, 128, 64);
const visResto = new Uint8Array(bufferMemoria, 192);
```

O tipo de visualização que usamos aqui é o `Uint8Array`, que diz que os tipos de dados do *buffer* da memória são inteiros de 8 bits não assinados, ou seja, valores de 0 a 255, equivalente ao tipo `uint8_t` em C. O construtor da visualização recebe o *buffer* da memória e opcionalmente o deslocamento inicial e o número de elementos.

Assim que um `ArrayBuffer` é criado, a única operação possível é obter o seu respectivo tamanho usando a propriedade `byteLength`, assim como no exemplo a seguir:

```
const bufferMemoria = new ArrayBuffer(16);
console.log(buffer.byteLength); // 16 bytes
```

Nesse código, criamos um *buffer* de memória com 16 bytes que são inicializados com o valor "0". O tamanho da memória afeta a forma com que a visualização mapeará os dados. Isso significa que uma memória de 16 bytes (128 bits) usando a visualização `Uint8Array` é vista para cada 8 bits (2 bytes), resultando em 16 "blocos" de memória. Por outro lado, uma visualização de 64 bits usando uma memória que contém o tamanho total de 128 bits é basicamente dividida em dois "blocos".

A imagem a seguir, retirada da documentação da Mozilla sobre matrizes tipadas (disponível em https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays), demonstra os valores de um *ArrayBuffer* de 16 bytes aplicados em diferentes visualizações:

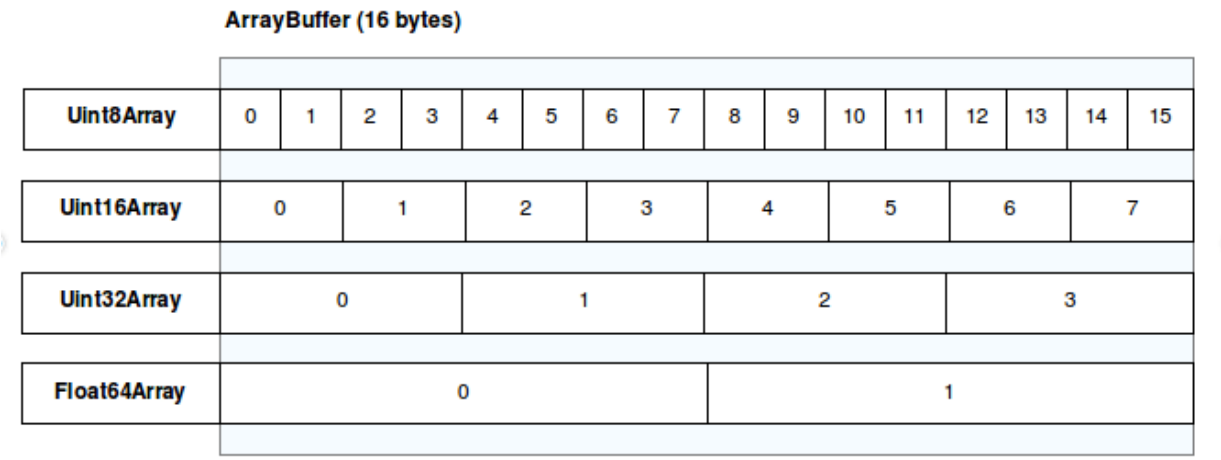


Figura 3.1: Typed Array.

Existem visualizações para todos os tipos numéricos comuns, com nomes autodescritivos como `Float32Array`, `Float64Array`, `Int32Array`, `Uint8Array` e outros. Há também uma visualização especial que substituiu o tipo de matriz de *pixels* na interface "ImageData" do HTML5 Canvas: a `Uint8ClampedArray`.

Muito legal, né? Tudo isso abre muitas possibilidades. Por exemplo, considere a `struct` a seguir, escrita em Rust, que pode vir a ser compilada para um arquivo binário depois:

> Nota: o alinhamento da estrutura de dados em uma estrutura Rust depende da plataforma e configuração usada. Essas diferenças de preenchimento podem mudar a forma como você lida com a visualização depois no JavaScript.

```
struct Pessoa {
  // Tamanho: 8 bits, 1 byte
  idade: u8
  // Nome é um array de 8 elementos que contem valores entre 0 e 255
  // Tamanho: 64 bits, 8 bytes
  nome: [u8; 8],
}
```

Conhecendo o tamanho e o mapeamento correto dos dados salvos em memória, posteriormente seria possível acessar usando:

```
// 64 bits + 8 bits = 72 bits = 9 bytes
const memoria = new ArrayBuffer(9);
```

Nesse código, apenas alocamos o espaço em memória, porém seria necessário salvar os dados da *struct* `Pessoa` no ArrayBuffer. Vamos ver no próximo capítulo como funciona essa operação, mas, por ora, imagine que os dados brutos estão salvos no *buffer* e o conteúdo não é apenas de valores "0". Seguindo o exemplo, podemos acessar os respectivos dados com o seguinte código:

```
// O valor "0" é deslocamento inicial e "1" é número de elementos.
// Dado que a idade é escrita em 8 bits é basicamente um "bloco" na
memória.
const visualizacaoIdade = new Uint8Array(memoria, 0, 1);
// Retorna o valor da idade
const idade = visualizacaoIdade[0];

// O valor "1" é deslocamento inicial e não especificamos
// o número de elementos, pois queremos o resto da memória
const visualizacaoNome = new Uint8Array(memoria, 1);
```

O bloco de código anterior permite criar uma visualização simples para um *buffer* de memória. A visualização em si já diz explicitamente quais os tipos de dados existentes dentro da memória, tornando simples a sua manipulação.

> Nota: a visualização do nome retorna uma lista de valores numéricos entre 0 e 255. Para transformar esses valores em *Strings*, é necessário recorrer à decodificação. Vamos ver como funcionam a codificação e decodificação de dados no capítulo 6 deste livro.

Outra forma de criar matrizes tipadas é colocar diretamente seu tamanho no construtor como inteiro. No código a seguir,

exemplificamos diferentes visualizações usando o valor `64`:

```
// Matrizes tipadas de números flutuantes
const f64 = new Float64Array(64);
const f32 = new Float32Array(64);

// Matrizes tipadas de números inteiros
const u32 = new Uint32Array(64);
const u16 = new Uint16Array(64);
const u8 = new Uint8Array(64);
const pixels = new Uint8ClampedArray(64);

// Matrizes tipadas de números inteiros assinalados
const i32 = new Int32Array(64);
const i16 = new Int16Array(64);
const i8 = new Int8Array(64);
```

Perceba que cada `ArrayBuffer` criado pelo construtor da visualização terá tamanhos diferentes de bytes; o uso é diferente de passar um *buffer* da memória, pois o comprimento não muda baseado na visualização usada.

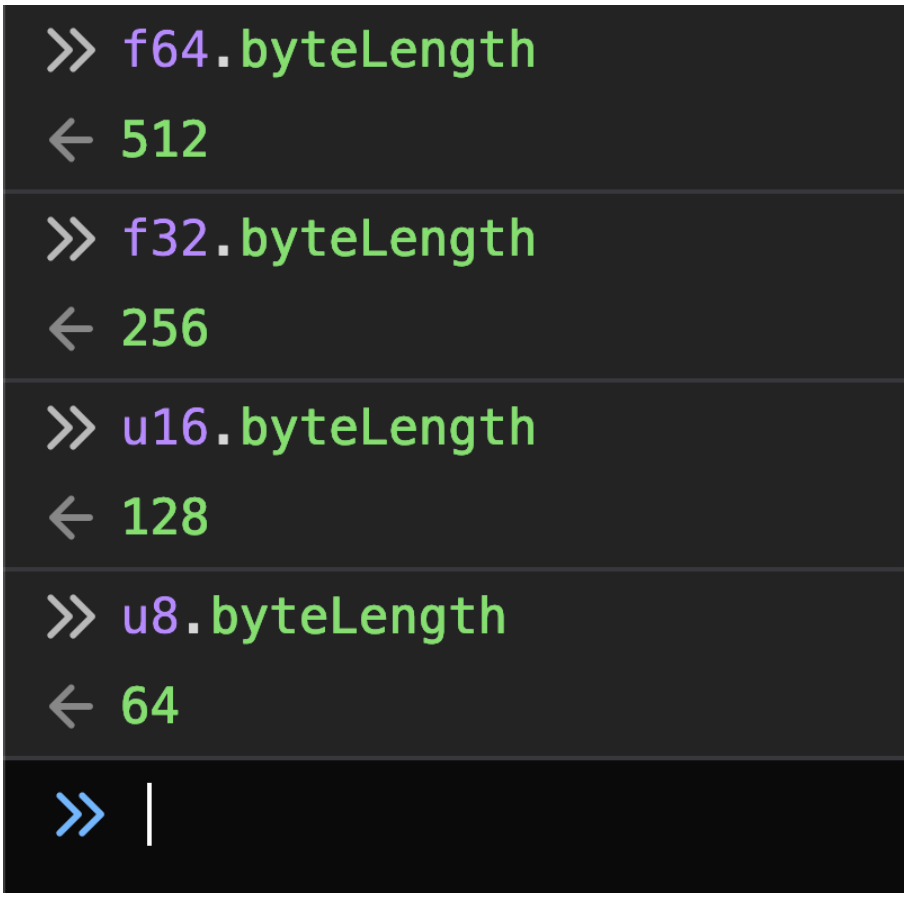


Figura 3.2: Tamanhos de bytes.

Essa forma é bem prática em comparação com a anterior, pois pula a criação declarativa de um `ArrayBuffer`; entretanto, opera a criação de um `ArrayBuffer` com base na visualização usada. É muito importante entender sobre `ArrayBuffer`, uma vez que usamos esse valor por baixo dos panos em qualquer matriz tipada, além de ser usado explicitamente pela API do WebAssembly.

Além das matrizes tipadas, existe outra forma de trabalhar com *buffers* de memória, conhecida como `DataView`, que se destina a lidar com dados heterogêneos.

3.2 DataView

Em vez de ter uma API semelhante a uma matriz, o objeto `DataView` fornece uma API parecida com um `Map` do JavaScript (`HashMap` no Rust) para ler e gravar tipos de dados em deslocamentos de bytes arbitrários. `DataView` funciona muito bem para ler e gravar cabeçalhos de arquivo e outros dados semelhantes à *struct* que escrevemos anteriormente (`Pessoa`).

O `DataView` ignora a forma como a plataforma lida com a "ordem de bytes" (termo conhecido como *endianness* em inglês). Suponha que tenhamos um formato de arquivo que tenha um cabeçalho com um número inteiro sem sinal de 8 bits seguido por um inteiro de 16 bits e uma matriz de números flutuantes de 32 bits. Ler isso com matrizes tipadas é possível, mas um pouco trabalhoso. Neste mesmo exemplo, usando um `DataView` , podemos ler o cabeçalho e usar uma exibição de matriz digitada para a matriz flutuante.

```
const dataView = new DataView(meuArrayBuffer);

// Primeiro byte do ArrayBuffer
const cabecalho = dataView.getUint8(0);

// 8 bits = 1 byte, então apontamos para o segundo byte
// Unit16 são 2 bytes, então a próxima endereço disponível é "3"
const comprimento = dataView.getUint16(1);

// Cria a visualização de dados de números flutuantes de 32 bits
// Com o tamanho baseado no valor do cabeçalho e comprimento
const matrizTipada = new Float32Array(comprimento * cabecalho);

// Cria a matriz tipada movendo de 4 em 4 bytes
for (let i = 0, endereco = 3; i < matrizTipada.length; i++, endereco += 4)
{
  matrizTipada[i] = dataView.getFloat32(endereco);
}
```

Outro recurso útil do `DataView` é que você pode controlar a ordenação de bytes (*endianness*) usando verdadeiro ou falso nos métodos do `DataView` . A ordenação de bytes é baseada em ordem de

grandeza e a ordem mais comum de bytes é conhecida como *little endian*, que é usada em todos os processadores Intel e AMD. A ordenação *little endian* armazena bytes na ordem do menor primeiro. A outra ordenação existente, conhecida como *big endian*, é justamente o oposto: armazena os bytes na ordenação maior primeiro. Essa ordenação é usada por alguns processadores como Motorola 68000, IA-64 e outros, além de ser usada pelo protocolo TCP/IP.

A fim de exemplificar as duas ordenações, se desejássemos salvar o valor numérico `305419896` de 4 bytes na memória, a forma como a plataforma lida com a organização de bytes faria a diferença na ordem de bytes salvos. A conversão do valor `305419896` para um hexadecimal obtém `0x12345678`.

Note que, para cada 2 caracteres do formato hexadecimal, temos o equivalente a 1 byte; logo, se o valor numérico `305419896` representado de forma hexadecimal (`0x12345678`) fosse salvo na memória usando *little endian*, teríamos a seguinte ordem:

- Em hexadecimal: `78 56 34 12` ;
- Em bytes: `0x78 0x56 0x34 0x12` .

Com a ordenação *big endian*, obteríamos a seguinte ordem:

- Em hexadecimal: `12 34 56 78` ;
- Em bytes: `0x12 0x34 0x56 0x78` .

Curiosidade: existe também outra ordenação conhecida como *middle-endian* (também chamada de "*mixed-endian*"), mas ela é pouco utilizada atualmente e é vista por muitos como algo histórico. Se você tiver interesse em aprender mais sobre ordenação de bytes, existe um livro excelente, porém escrito em inglês: *Mastering Reverse Engineering* (em português, *Dominando a Engenharia Reversa*), do Reginald Wong.

Na prática, você pode especificar qual ordenação você deseja usar nos métodos do `DataView` com o uso de um tipo `bool`. Se o valor for verdadeiro, será aplicada a ordenação *little endian*; do contrário, será aplicada a ordenação *big endian*.

```
const valorComLittleEndian = dataview.getUint32(0, true);
const valorComBigEndian = dataview.getUint32(4, false);
```

Você também pode verificar qual ordenação você está usando em sua respectiva plataforma com a seguinte função, que retorna verdadeiro se a ordenação for *little endian*:

```
function ordenacaoLittleEndian() {
  const buffer = new ArrayBuffer(2);
  new DataView(buffer).setInt16(0, 256, true);
  return new Int16Array(buffer)[0] === 256;
};

console.log(ordenacaoLittleEndian()); // true
```

Entender sobre como funciona a ordenação de bytes e `DataView` permite criar aplicações WebAssembly que são consistentes para qualquer uso multiplataforma ou para casos onde você especificamente quer salvar bytes do maior para o menor — extremamente útil para transmissões de rede.

Por enquanto, isso é tudo que você precisa saber a respeito de leitura e escrita de dados brutos na Web usando matrizes tipadas ou até mesmo fazendo uso do `DataView`. Dito isso, já podemos ver como funciona a API do JavaScript.

3.3 WebAssembly no JavaScript

O JavaScript provê uma API para lidar com WebAssembly no navegador para realizar operações de carregamento, compilação e instanciamento. Futuramente, o WebAssembly poderá ser carregado

igual a módulos ES6, simplesmente adicionando o tipo `module` na *tag script* do HTML (exemplo: `<script type="module"/>`), entretanto ainda não é uma realidade.

Na imagem a seguir, você pode ver as funções disponíveis na API `WebAssembly` do navegador que está dentro do escopo *window*, que também pode ser acessado pelo uso de `window.WebAssembly`:

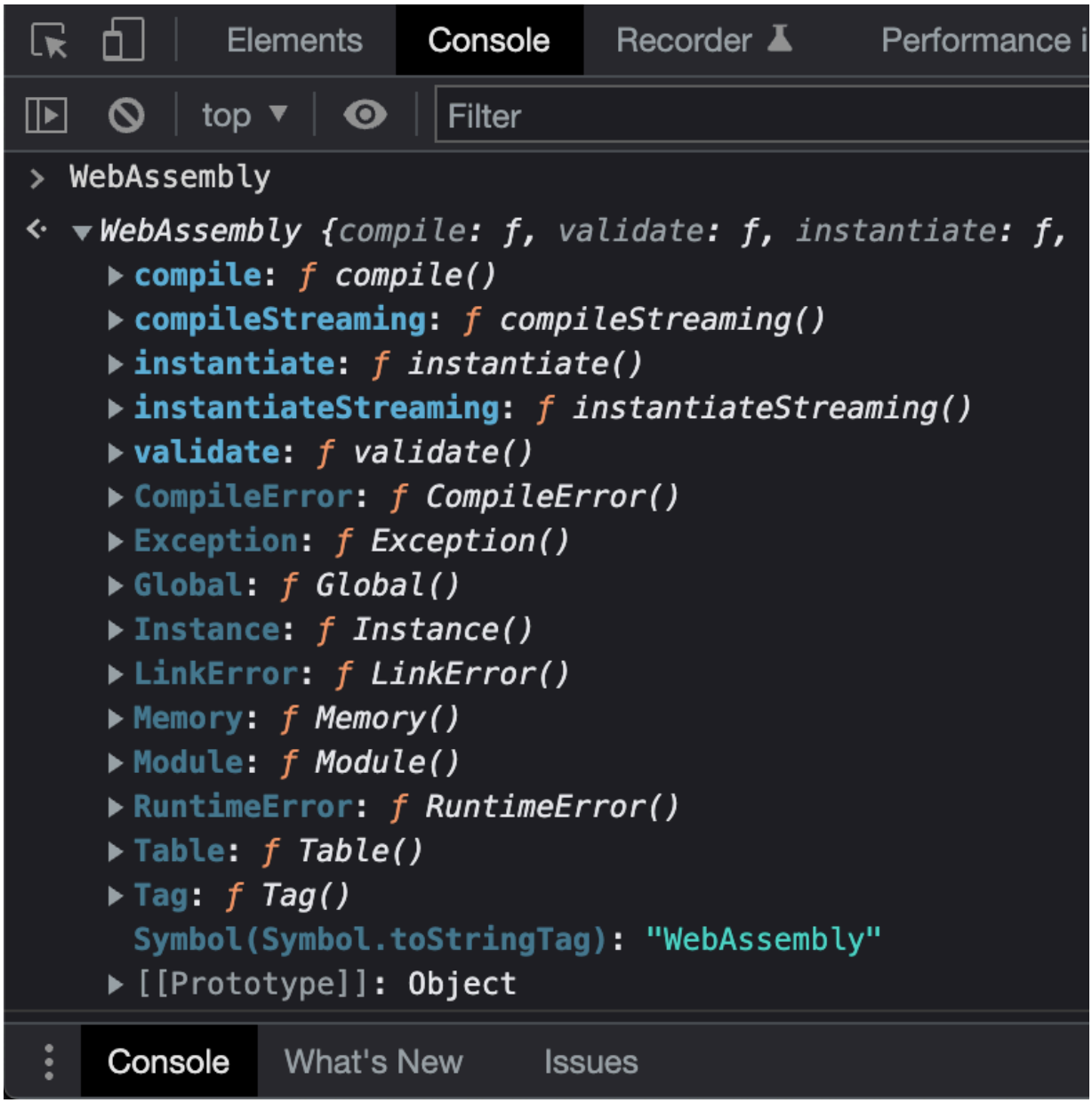


Figura 3.3: Funções disponíveis na API do JavaScript.

É importante notar que há classes no objeto do `window.WebAssembly` que também são conhecidas como "funções especiais" no JavaScript, motivo pelo qual todas as propriedades do objeto possuem um `f` do lado, que representa o tipo *função*. Essas classes são implementações de fundamentos do WebAssembly — algumas delas podem ser familiares, já que as vimos no capítulo anterior.

Vamos dar uma olhada em alguns métodos e classes da API que o navegador fornece, mas, neste capítulo, pularemos `Tag`, `CompilerError`, `RuntimeError`, `LinkError` e `Exception`, pois veremos como funciona o tratamento de erro e o *debug* de aplicações posteriormente no livro. A maioria das funções e classes mostradas na imagem giram em torno do *módulo* WebAssembly, e é justamente por isso que vamos começar nossa jornada pela classe `Module`.

3.4 Módulo

A API do WebAssembly disponibiliza uma classe `Module`, cujo construtor recebe os bytes do módulo. Essa classe pode ser instanciada inúmeras vezes e tem como retorno um código WebAssembly sem estado, que foi compilado pelo navegador.

A classe pode receber os bytes em forma de matriz tipada ou em um *ArrayBuffer*. No exemplo a seguir, temos os bytes do menor código binário válido para um módulo WASM:

```
const modulo = new WebAssembly.Module(
  Uint8Array.of(0x0, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00)
);

console.log(modulo); // WebAssembly.Module {  }
```

Nesse código, estamos literalmente compilando WebAssembly diretamente no navegador. É importante destacar que você pode

criar módulos sem necessariamente precisar criar sua instância, sendo possível ter inúmeros módulos criados e nenhuma instância.

```
// retorna a lista de importações no arquivo binário
console.log(WebAssembly.Module.imports(modulo));

// retorna a lista de exportações no arquivo binário
console.log(WebAssembly.Module.exports(modulo));
```

Se os bytes usados na criação do módulo estiverem incorretos, o JavaScript vai disparar um erro. A mensagem de erro geralmente dependente de caso para caso, entretanto é bem comum ver erros relacionados ao *número mágico* do binário, como estes:

```
Uncaught CompileError: at offset 4: failed to match magic number

0000004: error: bad magic value
```

Esses erros são formas de dizer que os primeiros bytes estão incorretos, logo o arquivo `.wasm` é inválido para uso. A API JavaScript disponibiliza uma função para lidar com esse tipo de caso chamada `validate` ("validar", em inglês). A função `validate` é um método simples que valida se uma matriz tipada ou um *ArrayBuffer* possui o conjunto de bytes de acordo com a especificação de um módulo WASM válido no formato binário. O retorno do método é verdadeiro ou falso (`true` ou `false`), dependendo da operação de validação.

A seguir, veja um exemplo de uso usando o método `arrayBuffer` da interface `Response` , disponibilizada pela API de *fetch* do navegador:

```
fetch('arquivo-com-bytes-invalidos.wasm')
  .then(response => response.arrayBuffer())
  .then(bytes => {
    // Retorna "false"
    const valido = WebAssembly.validate(bytes);
    if (!valido) {
      console.log('os bytes são inválidos');
    } else {
```

```
      const modulo = new WebAssembly.Module(bytes);
      console.log('os bytes são válidos');
      console.log('módulo criado:', modulo);
    }
  }
```

No caso anterior, quando usamos o método `validate`, estamos esperamos que o cenário de erro possa existir; logo, nenhum erro em tempo de execução no lado do JavaScript será lançado se tratarmos corretamente quando os bytes são inválidos.

Existem outras duas formas de compilação de módulo além de passar diretamente os bytes no construtor do `WebAssembly.Module`, como fizemos anteriormente, e ambas as opções de compilação funcionam como funções que podem ser chamadas diretamente do objeto `WebAssembly`. A primeira função é conhecida como `compile`, que compila os bytes em um objeto `WebAssembly.Module`. Essa função é baseada em JavaScript `Promises`.

Reescrevendo o exemplo anterior usando o método `compile`:

```
const bytes = Uint8Array.of(0x0, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00,
0x00);

if (WebAssembly.validate(bytes)) {
  const compilarBytes = WebAssembly.compile(bytes);
  compilarBytes.then((modulo) => {
    // WebAssembly.Module {  }
    console.log(modulo);
  });
}
```

A segunda forma alternativa é fazer uso da função `compileStreaming`, que faz literalmente a mesma coisa que a função `compile`, porém permite passar como argumento um objeto `Response` ou uma `Promise` referente ao módulo que você quer compilar por *stream*, sendo extremamente útil para quando queremos compilar módulos que precisam ser carregados.

A seguir, temos um exemplo de uso do método `compileStreaming` usando um arquivo binário imaginário chamado *meu-modulo-imaginario.wasm*:

```
// Módulo compilado porém não foi instanciado
const compilarModulo = WebAssembly.compileStreaming(
  fetch('meu-modulo-imaginario.wasm')
);
```

A função `compileStreaming` é amplamente usada pela simplicidade de escrita quando comparada com a opção de uso da função `compile` ou mesmo usando os bytes no construtor do `WebAssembly.Module`.

Um módulo compilado tem um valor limitado de certa forma, pois não podemos acessar a sua memória, nem ao menos executar as funções. Para fazer uso de um módulo WebAssembly no navegador de forma completa, precisamos ir para o próximo passo, que é criar a instância do módulo.

3.5 Instanciamento de módulos

Instâncias são representações executáveis e com estado de um módulo WebAssembly. A instância contém os dados da memória e todas as funções exportadas após compilação de um binário.

Assim como o módulo possui sua "função especial" que atua como classe, a instância também a possui, conhecida como `WebAssembly.Instance`. O código a seguir exemplifica a criação de uma instância a partir de um módulo compilado no navegador:

```
const bytes = Uint8Array.of(0x0, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00,
0x00);

if (WebAssembly.validate(bytes)) {
  // WebAssembly.Module {  }
  const modulo = WebAssembly.Module(bytes);
```

```
  // WebAssembly.Instance {  }
  const instancia = WebAssembly.Instance(modulo);
}
```

Esse código vai criar uma instância do módulo que foi compilado anteriormente. O uso do construtor em `WebAssembly.Instance` não é a única forma de criar instâncias. Assim como há opções alternativas para compilação e criação de módulo, existem duas outras formas para o processo de instanciamento.

A primeira opção alternativa é fazer uso do método `instantiate`, como no exemplo a seguir:

```
const compilarModulo = WebAssembly.compileStreaming(
  fetch('meu-modulo-imaginario.wasm')
);

compilarModulo
  .then(modulo => WebAssembly.instantiate(modulo));
```

O `instantiate` recebe o módulo já compilado e cria uma instância do módulo WebAssembly. Entretanto, é considerado atualmente uma forma ineficiente de instanciamento de módulos e, muitas vezes, seu uso não é recomendado justamente pela existência do método `instantiateStreaming`.

O método `instantiateStreaming` é bem parecido com `compileStreaming`, pois nele também podemos passar o `Response` do `fetch`. Assim como o método `compileStreaming`, ele realiza a compilação e criação do módulo, porém faz a operação adicional de instanciamento.

Veja um exemplo usando `instantiateStreaming` para compilação, criação do módulo e instanciamento:

```
WebAssembly.instantiateStreaming(
    fetch('meu-modulo-imaginario.wasm')
  )
  .then(resultado => {
    // Instância do módulo criada
```

```
    console.log(resultado.instance);
  });
```

Em resumo, `instantiateStreaming` busca, compila e cria uma instância de um módulo em uma única etapa, diretamente do código binário bruto, para que não exija conversão para um `ArrayBuffer`. O tamanho do código em si acaba se tornando mais curto e esta abordagem é considerada a maneira mais eficiente e otimizada de carregar o código WASM.

> Importante: para que funcione corretamente o uso do `instantiateStreaming`, os arquivos `.wasm` devem ser retornados com o tipo de mídia (*MIME*) `application/wasm` pelo servidor.

Os métodos `instantiate` e `instantiateStreaming` também permitem que você passe um segundo parâmetro. Esse segundo parâmetro é um objeto que contém os valores que serão importados para a instância recém-criada. Esses valores podem ser, por exemplo, funções ou objetos `WebAssembly.Memory`.

Para ficar mais claro, vamos ver um exemplo de como exportar funções. Imagine que, no nosso arquivo binário imaginário, que usamos até então nos exemplos de instanciamento, executamos uma função exportada chamada `incrementar` que sempre retorna o valor usado com a adição do valor 1 — logo, se usarmos `85` como parâmetro, será retornado `86`.

Posteriormente vamos aprender como as importações funcionam de forma prática, mas, por ora, imagine o seguinte cenário: dentro do código da função `incrementar`, querermos executar o `console.log` do JavaScript assim que recebermos o valor. Veja a seguir a lógica descrita em um pseudocódigo Rust:

```
pub fn incrementar(valor: i32) -> i32 {
    console_log_do_javascript(valor);
    let novo_valor = valor + 1;
```

```
    novo_valor
}
```

Neste cenário, seria necessário o usar o objeto de importação e mapear a função `console_log_do_javascript` tanto no lado do navegador, quanto no WebAssembly. A seguir, temos o caso descrito com um exemplo:

```
// Objeto que contém informações das importações
const objeto = {
  imports: {
    // Mapeia console_log_do_javascript para a instância
    console_log_do_javascript: valor => console.log(valor)
  }
};

WebAssembly
  .instantiateStreaming(fetch('meu-modulo-imaginario.wasm'), objeto)
  .then(resultado => {
    // No momento da execução da função,
    // console_log será chamado e exibirá o valor "85"
    const novoValor = resultado.instance.exports.incrementar(85);
    // Exibe "86"
    console.log(novoValor);
  });
```

Entretanto, não é sempre que você precisa ou quer definir as funções a serem importadas pelo WebAssembly. Isso depende de como você escreveu o módulo e suas respectivas instâncias. No mesmo objeto de importações, podemos definir também a configuração de memória.

3.6 Criando e gerenciando memória

A memória no WebAssembly é criada a partir da classe `Memory` e seu retorno é um objeto que contém um método e uma propriedade,

que pode ser um `ArrayBuffer` ou `SharedArrayBuffer` (é basicamente um `ArrayBuffer`, mas que pode criar visualizações com memória compartilhada) que contém os bytes brutos da memória acessada por uma instância WebAssembly.

Toda vez que você executa o construtor da classe `Memory`, cria-se um objeto de memória que possui apenas o método `grow` ("crescer", em inglês) e a propriedade `buffer`, que contém um `ArrayBuffer` ou `SharedArrayBuffer`.

No exemplo de código a seguir, criamos uma memória configurada inicialmente com o uso de uma página de memória (64 *kibibytes*, KiB) e com o uso máximo de 100 páginas (6.4 *mebibyte*, MiB):

```
const memoria = new WebAssembly.Memory({
  initial: 1,
  maximum: 100
});

console.log(memoria.buffer); // ArrayBuffer { byteLength: 65536 }
```

Temos um `ArrayBuffer` criado a partir das configurações que usamos no construtor. Você também pode especificar que deseja criar uma memória compartilhada usando a propriedade `shared` como verdadeira. Quando a memória é compartilhada, o seu respectivo valor é um `SharedArrayBuffer` em vez de `ArrayBuffer`. Veja o exemplo:

```
const memoria = new WebAssembly.Memory({
  initial: 1,
  maximum: 100,
  shared: true
});

console.log(memoria.buffer); // SharedArrayBuffer { byteLength: 65536 }
```

Podemos configurar o uso de memória no WebAssembly no objeto que usamos anteriormente para mapear as importações. No exemplo a seguir, usamos o método `instantiateStreaming` para compilação e instanciamento do módulo com uma configuração de memória:

```
const memoria = new WebAssembly.Memory({ initial: 10, maximum: 100 });

const objetoDeImportacao = {
  env: {
  mem: memoria
  }
};

WebAssembly.instantiateStreaming(fetch('meu-modulo-imaginario.wasm'),
objetoDeImportacao);
```

Perceba que a memória é definida dentro de um objeto na chave `env`, que configura instruções do JavaScript. Outro detalhe interessante é que a chave que é a entrada para configuração da memória é nomeada como `mem`. Os nomes usados nas chaves (`env` e `mem`) não são necessariamente um padrão, pois dependem dos nomes das importações configurados no compilador da linguagem.

A chave `env` é o padrão atual do destino de compilação do WebAssembly usando Rust; entretanto, é possível modificar ambas as nomenclaturas e usar a chave que você achar melhor.

Ainda a respeito do exemplo anterior, a configuração da memória máxima pode ser aumentada com o uso do método `grow`. A memória foi configurada com o valor inicial de 10 páginas (640 *kibibytes*), então quando executarmos o `grow`, ela mudará de 10 páginas para o valor especificado. Podemos definir quantas páginas queremos adicionar como argumento do método.

Kibibyte é um número usado para medir o espaço de armazenamento nos discos rígidos ou na memória do computador. O *Kibibyte* é sempre composto por 1.024 bytes e é frequentemente comparado com o *Kilobyte*. O *Kilobyte* pode ser representado por 1.000 ou 1.024 bytes, dependendo da especificação utilizada. De acordo com o IEC (International Electrotechnical Commission), o nome correto para 1.024 bytes é *Kibibyte*.

No código a seguir, temos um cenário que exemplifica um elemento na página HTML com um identificador (*id*) equivalente a `botao`, que, quando clicado, aumenta a memória em uma página:

```
WebAssembly.instantiateStreaming(fetch('meu-modulo-imaginario.wasm'),
importacoes)
  .then(resultado => {
  const memoria = resultado.instancia.exports.memory;
  const botao = document.getElementById('botao');
  botao.addEventListener('click', () => {
    try {
      // Aumenta a memória em 1 página
      memoria.grow(1);
    } catch() {
      console.log('Você não pode aumentar mais a memória');
    };
  }, false);
  });
```

Nesse código, repare que temos um `try` e `catch` em volta da lógica responsável por aumentar a memória. Infelizmente, não há nenhum método disponível para validar operações de memória no WebAssembly e, dado que nem sempre é possível aumentar a memória do módulo, erros em tempo de execução podem ocorrer.

Assim como a memória, o WebAssembly também disponibiliza a criação de valores globais que podem ser compartilhados de forma declarativa.

3.7 Valores globais

A classe `Global` da API do WASM no navegador permite criar uma variável global, que é acessível no escopo do JavaScript, como também importável e exportável em uma ou mais instâncias de um módulo.

Seu uso é bem simples: você define o tipo de valor e se é mutável ou não. O exemplo de código a seguir mostra um valor global do tipo `i32` criado usando o construtor da classe `Global`. Nos parâmetros do construtor, você usa o objeto de configuração do global e seu respectivo valor.

```
const configuracao = {
  // Tipo do valor, no caso um 'i32'
  value: 'i32',
  // Se o valor é mutável ou não
  mutable: true
}

const valorInicial = 85;

// Cria a variável global mutável
const globalMutavel = new WebAssembly.Global(configuracao, valorInicial);
```

Após definirmos o valor global, podemos reutilizar a variável em uma instância (ou mais). O código a seguir exemplifica como o global pode ser definido no objeto de importações dos métodos de instanciamento:

```
const importacoes = {
  env: {
  global: globalMutavel
  }
};

WebAssembly.instantiateStreaming(fetch('global.wasm'), importacoes)
  // ...
```

Vale lembrar que o valor global pode ser modificado pelo lado do JavaScript ou pelo WebAssembly. No JavaScript, seria simplesmente mudar o valor em sua referência.

Agora que entendemos como as importações podem ser configuradas usando a API do WebAssembly no JavaScript, vamos ao último assunto do capítulo: tabelas.

3.8 Tabelas

As tabelas são importantes para aplicações WebAssembly pelo suporte provido à execução de funções de forma dinâmica. Vamos entender melhor sobre tabelas especialmente no capítulo sobre o formato de texto do WebAssembly, mas, por ora, vamos ver de forma breve como elas funcionam.

Tabelas funcionam basicamente como listas que contêm referências de funções. Para uma melhor visualização, vamos imaginar uma tabela nomeada como `tabela`, que possui duas funções que, respectivamente, retornam 23 e 85. Escrever esse exemplo de forma simplória usando JavaScript seria algo similar ao seguinte código:

```
function primeiraFuncao() {
  console.log(23);
}

function segundaFuncao() {
  console.log(85);
}

const tabela = [
  primeiraFuncao,
  segundaFuncao
];
```

```
tabela[0](); // retorna 23
tabela[1](); // retorna 85
```

Tabelas são extremamente úteis, pois podemos chamar funções baseadas em índices e, com isso criar cadeias de operações baseadas em valores dinâmicos obtidos em tempo de execução, algo que não é possível no WebAssembly sem o uso de tabelas.

A API do WebAssembly no navegador disponibiliza uma classe chamada `Table`, que permite criar e gerenciar as referências de funções no navegador, sendo possível configurar a tabela quando for instanciar um módulo, definindo o tamanho inicial da tabela e seu respectivo nome.

Uma tabela criada pelo JavaScript ou no código WebAssembly é acessível e mutável por ambos. A definição da tabela no objeto de importações segue o mesmo padrão da forma que fizemos com a memória e valores globais. Entretanto, é importante destacar que a tabela é modificável e permite adicionar ou remover itens de sua coleção após a sua iniciação. O código a seguir demonstra a definição da tabela no objeto de importações usando `instantiateStreaming`:

```
const tabela = new WebAssembly.Table({
  initial: 2,
  element: "tabela"
});

const importacoes = {
  env: { tabela }
};

// Retorna "null"
console.log(tabela.get(0));
// Retorna "null"
console.log(tabela.get(1));

WebAssembly.instantiateStreaming(
  fetch('meu-modulo-imaginario.wasm'), importacoes
```

```
).then((resultado) => {
  // Retorna 2
  console.log(tabela.length);
  // Retorna 23
  console.log(tabela.get(0)());
  // Retorna 85
  console.log(tabela.get(1)());
});
```

Nesse código, repare que definimos um tamanho inicial e o nome da tabela. Podemos notar também que, até o módulo ser instanciado, os itens da tabela vão retornar `null`.

Perfeito! Por ora, já sabemos tudo que precisamos saber sobre como a API do WebAssembly funciona. Vimos como compilar e instanciar módulos, além de ver sobre a criação de tabelas, memória e definições de importações. Mas chega de módulos imaginários, vamos colocar a mão na massa no próximo capítulo e colocar todos esses conhecimentos em prática para escrever um editor de imagens com Rust compilando para WebAssembly.

Capítulo 4
Criando um editor de imagem com WebAssembly

Que tal usarmos tudo que aprendemos para criar um editor de imagem com Rust, JavaScript e WebAssembly?

Aplicações que realizam processamento de imagens geralmente são boas candidatas ao uso do WebAssembly pela natureza de uma função de processamento, que se baseia em aplicar mudanças de dados de acordo com a regra da manipulação especificada.

Nos últimos capítulos, pudemos aprender sobre:

- Os fundamentos básicos do WebAssembly (módulos, instâncias, memória, globais);
- Tipos de dados suportados;
- As diferenças entre o formato de arquivo binário e arquivo de texto;
- A API do JavaScript, que é usada para carregar, compilar e instanciar módulos no navegador.

Vamos colocar todos esses aprendizados em prática neste capítulo. Começaremos com o uso do `cargo`, o gerenciador de pacotes do Rust, para gerar o nosso editor em forma de pacote seguindo o *template* (modelo padrão) do `cargo`.

> **Lembrete:** se você tiver curiosidade ou precisar verificar algo, todos os exemplos do livro (incluindo os arquivos binários) estão neste link: https://github.com/raphamorim/livro-webassembly-exemplos.
>
> Basta clonar o repositório usando git ou fazer uso da opção de download do código-fonte que o GitHub provê.

O `cargo` permite criar diferentes tipos de pacotes. No caso do editor, queremos obter o pacote em formato de biblioteca e, para isso, precisamos especificar o uso da opção `--lib`. Não é necessário criar como uma biblioteca, você também pode criar um executável Rust usando o comando `cargo init editor --bin`. Observe, no entanto, que você precisa ter uma função `main()` com a assinatura bem estabelecida ou fechar o compilador usando a notação `#![no_main]`, para que ele saiba que a ausência da função `main()` é intencional.

Na maioria das vezes, os módulos WebAssembly assumem mais o papel de uma biblioteca do que o papel de um executável — exceto no contexto do WASI, que veremos posteriormente no livro. A abordagem da biblioteca é semanticamente preferível para o caso do editor, mas pode mudar de caso para caso. Você pode ver a lista completa de opções acessando a documentação oficial: https://doc.rust-lang.org/cargo/commands/cargo-new.html.

Execute o comando do `cargo` para criar um novo projeto, que será nomeado como `editor`:

```
cargo new editor --lib
```

Quando o comando terminar de ser executado, ele deve criar dois arquivos, `src/lib.rs` e `Cargo.toml`, como mostra a imagem a seguir:

Figura 4.1: Exemplo das pastas da aplicação.

Da mesma forma como fizemos no capítulo 2, vamos adicionar o `crate-type` como `cdylib`. É importante destacar que essa configuração especifica a produção de uma biblioteca de sistema dinâmica e é geralmente usada para construir bibliotecas que podem ser vinculadas a programas C e C++.

O arquivo `Cargo.toml` deve ser igual ao seguinte código:

```
[package]
name = "editor"
version = "0.1.0"
edition = "2021"

[lib]
crate-type = ["cdylib"]
```

A novidade deste capítulo é que vamos pular o uso do `wasm-pack` e `wasm-bindgen`, que usamos no capítulo 2. Desta vez, vamos escrever a lógica de carregamento e instanciamento de módulos WASM no JavaScript, dado que, no capítulo anterior, aprendemos a usar a API do WebAssembly no navegador.

Para isso, vamos precisar especificar o destino de saída como WebAssembly para o compilador (`rustc`). Você pode criar uma pasta na raiz do projeto nomeada como `.cargo` (intencionalmente com o ponto final na frente) e adicionar um arquivo de texto nomeado como `config`. Neste arquivo de texto, você pode especificar configurações do cargo para o projeto em questão.

O arquivo `config` na pasta `.cargo` é frequentemente usado para projetos que não têm mudanças no destino de saída ou possuem configurações bem específicas do compilador Rust. Já que não vamos produzir qualquer outro destino além de WebAssembly, podemos definir o destino de compilação padrão do nosso pacote `editor` como `wasm32-unknown-unknown`. No arquivo `config`, preencha com o seguinte código:

```
[build]
target = "wasm32-unknown-unknown"
```

É preciso ter o `wasm32-unknown-unknown` instalado. Fizemos juntos a instalação dele no capítulo 2, mas caso você ainda não o tenha instalado, basta executar no terminal:

```
rustup target add wasm32-unknown-unknown
```

Após isso, vamos atualizar nosso arquivo `lib.rs` que foi gerado automaticamente pelo `cargo` dentro da pasta `src`, pois queremos ver algo acontecendo até então no lado do navegador. Provavelmente o `cargo` criou um arquivo `lib.rs` com código existente juntamente com seu respectivo teste. Remova todo o conteúdo deste arquivo e vamos começar do zero, com a escrita de uma função de subtração entre valores `u8` (0 a 255):

```
fn subtracao(numero_a: u8, numero_b: u8) -> u8 {
    numero_a - numero_b
}
```

Nessa função, estamos recebendo dois valores `u8` e fazendo sua respectiva subtração. A função `subtracao` é suscetível a erros em tempo de execução, pelo fato de não ter nenhum tipo de controle a respeito do valor recebido pelo JavaScript, além de também ignorar resultados da subtração como sendo menor que o menor valor possível de um tipo `u8` (acessível pela constante `u8::MIN`) ou maior que o maior valor possível (`u8::MAX`). Neste capítulo, estamos considerando o melhor caso possível sempre e não vamos focar na parte de erros e solução de problemas, então não se prenda a resolver esse tipo de problema por ora.

Outro ponto em relação à função `subtracao` é que a forma como ela é definida não será compilada para o arquivo binário do WebAssembly. Naturalmente, muitos desenvolvedores Rust acabam tentando resolver o problema da compilação adicionando a palavra-chave `pub`, como no código a seguir:

```
pub fn subtracao(numero_a: u8, numero_b: u8) -> u8 {
    numero_a - numero_b
}
```

Infelizmente, apesar de fazer sentido logicamente o uso da palavra-chave `pub`, o compilador Rust vai ignorar o fato de ser uma função pública e também não adicionará no arquivo binário. O uso da palavra-chave `pub` para sinalizar funções a serem exportadas vem sendo amplamente discutido na comunidade do WebAssembly e Rust, mas ainda não é uma realidade (e talvez nunca seja).

Para incluir a função `subtracao` no arquivo binário, é preciso usar a notação `#[no_mangle]`:

```
#[no_mangle]
fn subtracao(numero_a: u8, numero_b: u8) -> u8 {
    numero_a - numero_b
}
```

Com isso, a função será incluída no binário assim como a versão pública da mesma função também seria incluída com o uso da notação `#[no_mangle]`:

```
#[no_mangle]
pub fn subtracao(numero_a: u8, numero_b: u8) -> u8 {
    numero_a - numero_b
}
```

O atributo `no_mangle` desativa o desmembramento de nomes do Rust, para que seja mais fácil vincular a biblioteca produzida — no nosso caso, especificamente, permite a vinculação entre o JavaScript e o WebAssembly.

Em um cenário de compilação genérico, sem a notação `#[no_mangle]`, o compilador renomeia essa função e fica mais difícil de usá-la em qualquer outro código externo, pois o nome da função mudará arbitrariamente. Veja um exemplo a seguir, com e sem `#[no_mangle]`, lembrando que o nome da função sem `#[no_mangle]` não é constante, então pode ter outra nomenclatura:

```rust
// A função "retorna_42" será nomeada como "retorna_42"
// pois explicitamente evitamos o desmembramento usando
// a notação "no_mangle"

#[no_mangle]
pub fn retorna_42() -> u8 {
    42
}

// Sem o uso da notação "no_mangle", a função "retorna_85" terá
// o nome gerado pelo próprio compilador. Um exemplo dos
// possíveis nomes gerados é: _ZN8nomenclat13retorna_85

pub fn retorna_85() -> u8 {
    85
}
```

Inúmeros membros da comunidade abertamente não aceitam o uso da notação `#[no_mangle]` como dependência do destino de compilação para WebAssembly, pelo fato de ser uma notação que não foi criada para lidar exclusivamente com WASM e sim para compilação de artefatos que serão vinculados posteriormente.

Alternativamente, você também pode usar outra notação chamada `export_name`. Essa notação raramente é necessária, mas você pode exportar uma função com um nome diferente do nome interno do Rust. Veja um exemplo no qual exportamos a função `subtracao` como `"outra_subtracao"`:

```rust
#[export_name = "outra_subtracao"]
pub fn subtracao(numero_a: u8, numero_b: u8) -> u8 {
    numero_a - numero_b
}
```

Após compilar essa função usando `cargo build --release`, dentro de `target`, perceba que existe uma pasta nomeada `wasm32-unknown-unknown`. Nesta pasta, você vai encontrar outra pasta nomeada como `release`, e ali estará o nosso arquivo do formato binário do WebAssembly nomeado com o mesmo nome do pacote: `editor.wasm`.

É importante notar que, se você executar o comando `cargo build` sem usar o parâmetro `--release`, o arquivo será gerado em um caminho de arquivos diferente: em vez da pasta `release`, será salvo na pasta `debug`. Outra nota que vale ser ressaltada é que estamos compilando Rust para WebAssembly com sucesso. No entanto, versões futuras do Rust podem produzir um módulo WebAssembly com assinaturas de funções completamente diferentes.

A maneira como os parâmetros da função (exemplo: um ponteiro para uma memória ou como um valor imediato) são passados de *callee* (chamadores de funções) para *callee* faz parte da definição da *Application Binary Interface*, ou ABI (veremos mais sobre ABI no próximo capítulo). O `rustc` usa o ABI do Rust por padrão, que não é estável. Para estabilizar essa situação, podemos definir explicitamente qual ABI queremos que o `rustc` use para uma função. Isso é feito usando a palavra-chave `extern`.

```
#[no_mangle]
pub extern fn subtracao(numero_a: u8, numero_b: u8) -> u8 {
    numero_a - numero_b
}
```

No próximo capítulo, vamos entender a fundo quando o uso da palavra-chave `extern` é importante; por ora, não vamos entrar em detalhes de como funciona a ABI dessa função.

Execute um `cargo build --release` com esse código para atualizar nosso arquivo binário. Após a geração do nosso binário `editor.wasm`, podemos afirmar que estamos resguardados de forma independente da versão do Rust. Dito isso, pausa no Rust e vamos voltar ao universo do navegador.

4.1 De volta ao navegador

Voltamos ao nosso querido amigo, o navegador. Aprendemos no último capítulo algumas funções para lidar com WebAssembly e, neste capítulo, colocaremos essas funções em uso. O primeiro passo é a criação do arquivo HTML que vai conter a estrutura dos elementos do editor de imagens.

Criaremos o arquivo HTML com o nome de `index.html` e, dentro dele, vamos ter também as importações de um arquivo CSS e um arquivo JavaScript, que escreveremos posteriormente. O conteúdo do arquivo `index.html` deverá ser similar ao código a seguir:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <title>Editor</title>
    <link rel="stylesheet" href="editor.css">

    <!-- Essa linha é completamente opcional, adicionada apenas para
evitar a mensagem de erro do navegador quanto ao "favicon.ico" -->
    <link rel="icon" type="image/png" href="-" />
</head>
<body>
    <div class="painel">
        <h1>Escolher imagem:</h1>
        <input type="file" accept="image/jpeg, image/png" />
        <h1><div id="performance"></div></h1>
        <h1>Filtros:</h1>
        <p><button id="remover">Remover filtros</button></p>
        <p><button id="preto-e-branco-js">Filtro preto e branco usando
JavaScript</button></p>
        <p><button id="preto-e-branco-wasm">Filtro preto e branco usando
WebAssembly</button></p>
    </div>

    <div class="area">
        <img id="imagem" height="400" src="rio-de-janeiro.jpg"
        />
    </div>
```

```
    <script src="editor.js"></script>
</body>
</html>
```

Basicamente, a estrutura da página é dividida em `"área"` e `"painel"`. A área é responsável pela visualização da imagem que estamos editando. No código anterior, usei uma imagem da Wikipédia, porém sinta-se à vontade para usar qualquer outra imagem, pois isso não vai alterar o resultado final. O único requisito é que a imagem esteja no mesmo servidor do `index.html`, por isso é recomendado baixar a imagem em vez de usar fontes externas.

> A imagem rio-de-janeiro.jpg foi retirada do link https://upload.wikimedia.org/wikipedia/commons/thumb/9/98/Cidade_Maravilhosa.jpg/2880px-Cidade_Maravilhosa.jpg. Você também pode baixar a imagem pelo link do repositório: https://github.com/raphamorim/livro-webassembly-exemplos.

O elemento da classe `"painel"` possui os controles de edição de imagem, como botões para carregar o filtro, e também um *input* de arquivo para trocar a imagem. Os botões terão grande parte em como as funções importadas do WebAssembly serão executadas no editor.

A nossa imagem está dentro de um elemento com a classe nomeada como `"area"`. A `"area"` não tem nenhuma funcionalidade específica além dos recurso visual como uma ideia de grade por trás da imagem. O elemento do tipo imagem, definido com o identificador `"imagem"`, assim como os botões, terá relevância quando estivermos lançando e executando funções do JavaScript e WebAssembly, pois queremos acessar e modificar os dados desta imagem baseado no filtro do botão acionado.

Perceba também que, além dos elementos de painel, botões, área e imagem, estamos importando dois arquivos no `index.html`:

`editor.css` e `editor.js`. O arquivo `editor.css` será responsável pelos estilos da página Web e o `editor.js` vai conter toda a lógica de instanciamento do WebAssembly, como também as funções dos botões de filtro.

Que tal começarmos escrevendo os estilos dessa página no `editor.css`? Assim, podemos ver visualmente como cada elemento será posicionado na página. O arquivo `editor.css` deverá ser escrito desta forma:

```
body {
    margin-top: 10em;
    font-family: 'Helvetica';
    background: #000000;
    color: white;
    background-size: 40px 40px;
    background-image:
        linear-gradient(to right, grey 1px, transparent 1px),
        linear-gradient(to bottom, grey 1px, transparent 1px);
}

.area {
    background: black;
    width: 60%;
    height: 600px;
    display: flex;
    justify-content: center;
    align-items: center;
    background-size: 10px 10px;
    background-image:
        linear-gradient(to right, grey 1px, transparent 1px),
        linear-gradient(to bottom, grey 1px, transparent 1px);
}

.painel {
    padding: 2em 1em;
    background: #c12664;
    float: right;
```

```
    width: 25%;
}
```

Vamos testar essas alterações? Lembra do servidor que usamos no capítulo 2? Vamos inicializá-lo na raiz do projeto (caso não se lembre como inicializar o servidor, segue a primeira opção da lista dada anteriormente: `cargo install cargo-server && cargo server --port 5000`).

Acesse `http://localhost:5000` no navegador e você deverá ter algo parecido com a seguinte imagem:

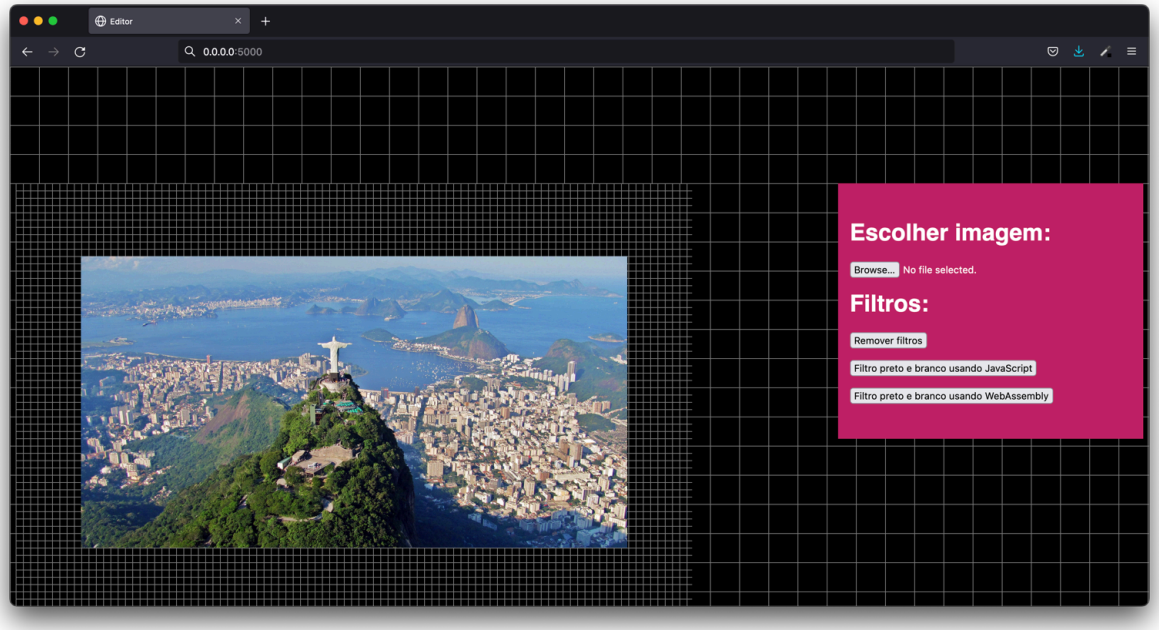


Figura 4.2: Estilo da página.

Perfeito! Temos os estilos sendo computados na página HTML da forma que esperamos, porém nenhum botão de filtro de imagem está funcionando. Dito isso, vamos escrever um pouco de JavaScript. Vamos carregar o arquivo `editor.wasm` produzido pelo código em Rust de subtração que escrevemos e compilamos para WebAssembly. Para isso, vamos criar o arquivo `editor.js` e preencher com o código a seguir:

```
// O caminho para o arquivo wasm da raiz do projeto
const arquivo = './target/wasm32-unknown-unknown/release/editor.wasm';

// Faz o download do arquivo e instancia o WebAssembly
WebAssembly.instantiateStreaming(fetch(arquivo))
  .then(wasm => {
    const { instance } = wasm;

    // A função "subtracao" está disponível no objeto "exports" da
instância
    const { subtracao } = instance.exports;
    console.log(subtracao(28, 10)); // 18
  });
```

Vamos ver se a subtração foi executada corretamente? Volte ao navegador e atualize o conteúdo da página. Após isso, acesse o *Console* e podemos ver que o código no navegador realizou com sucesso a operação de subtração:

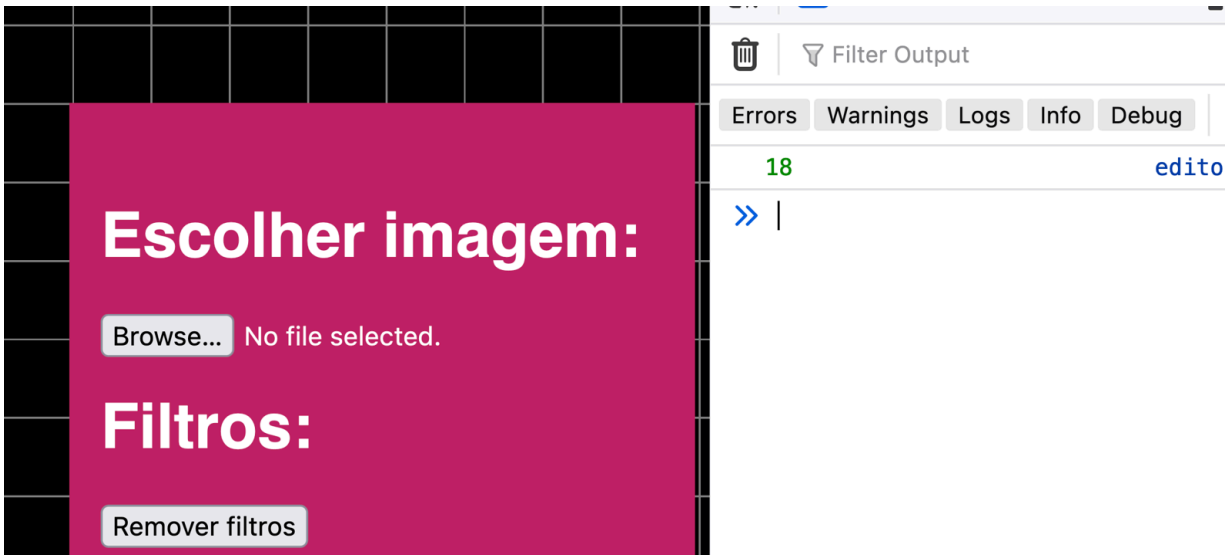


Figura 4.3: Resultado da função subtracao.

Bem legal saber que temos o carregamento do arquivo `editor.wasm` realizado pelo arquivo JavaScript de forma esperada, porém ele não faz nada além do que uma função de subtração. Queremos escrever

operações de processamento de imagem e é necessário usar conjuntos de dados maiores que apenas dois valores `u8`.

Os dados de uma imagens são basicamente listas de números e, dependendo do tipo da imagem, você tem uma lista de `u8`, `u16`, `u32` e outros formatos. Na Web, é bem comum o uso do tipo `u8` (especialmente manipulando imagens com a API do HTML5 Canvas, como faremos neste capítulo).

Logicamente, quanto mais pesada a imagem, maior o tamanho da lista. Considerando essa informação, trafegar e modificar dados de uma imagem pode ser uma operação cara em termos de processamento. O WebAssembly disponibiliza uma forma de armazenamento e manipulação de dados com o uso de sua memória. Na próxima seção, usaremos declarativamente a memória do WebAssembly para armazenar os dados da imagem.

4.2 Trabalhando com memória

O uso de memória do WebAssembly permite alocar dados bem maiores que podem ser trafegados e manipulados livremente no lado do Rust ou JavaScript. Nos blocos de memória, podemos reservar qualquer tipo de dados e informações que desejarmos, desde que façamos o uso correto do endereçamento e tamanho.

Nesta seção, veremos um pouco sobre como a memória funciona de forma prática. Escreveremos uma função responsável por criar a memória inicial chamada `criar_memoria_inicial` dentro do arquivo `src/lib.rs` e compilaremos usando o comando `cargo build --release`:

```
use core::slice::from_raw_parts_mut;

#[no_mangle]
extern fn criar_memoria_inicial() {
    let fatia: &mut [u8];
```

```
    unsafe {
        fatia = from_raw_parts_mut::<u8>(5 as *mut u8, 10);
    }

    fatia[0] = 85;
}
```

Nesse código, definimos uma fatia mutável do tipo `u8` e a nomeamos como `fatia`. Em seguida, usamos `from_raw_parts_mut` para criar um `u8` usando um ponteiro e o comprimento da memória. Por padrão, a memória começa do zero e pegamos apenas um elemento.

A função `criar_memoria_inicial` é anotada com `#[no_mangle]`, igual à função de `subtracao` que escrevemos no início do capítulo. Essa função não retorna nenhum valor, apenas altera o valor de um índice na memória.

A forma como escrevemos a função não é nada extravagante ou moderna, e sim uma aplicação bem direta do uso do `slice`, que é uma visão em um bloco de memória representado como um ponteiro e um comprimento. Existem outras formas de alocar a memória inicial, mas, pela simplicidade e didática, vamos manter o uso do `slice`.

> É importante frisar que a função `from_raw_parts_mut` pode não funcionar com ponteiros nulos como zero. Ponteiros nulos são disponíveis para uso quando a compilação é para `debug` em vez de `release`, pois o nível de otimização `debug` fornece essa possibilidade. Para mais informações sobre ponteiros nulos, você pode ver a discussão sobre o assunto no próprio repositório da linguagem Rust: https://github.com/rust-lang/rust/issues/53871.

Estamos acessando a memória bruta, então agrupamos as chamadas dentro do bloco não seguro (note o `unsafe` no código). A

princípio, nosso objetivo é fazer uma pequena alteração nesse bloco de memória para poder refletir no JavaScript.

Para realizar mudanças na memória, precisamos dos dados de forma mutável e, para obter a fatia mutável, usamos `from_raw_parts_mut` (`_mut` no final do nome do método identifica como mutável o seu retorno).

Já que queremos apenas ver uma pequena mudança, atribuímos `85` no primeiro índice do endereço `5` da memória. Podemos também reler o código escrito em Rust usando JavaScript para visualizar a memória sendo modificada pelo endereço `5` :

```
// A memória começa vazia, pois não foi inicializada,
// não se sabe o comprimento que ela possui, nem seu respectivo valor
let memoria = null;

// O comprimento da lista é 10
const comprimento = 10;

// Nosso ponteiro
const ponteiro = 5;

function criar_memoria_inicial(comprimento, ponteiro) {
  // A memória consiste em 10 itens nulos:
  // Array(10) [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
  memoria = new Array(comprimento).fill(0);

  // Modifica o quinto endereço para 85
  memoria[ponteiro] = 85;
}

criar_memoria_inicial(comprimento, ponteiro);

// Array(10) [ 0, 0, 0, 0, 0, 85, 0, 0, 0, 0 ]
console.log(memoria);
```

No JavaScript, quando chamamos a função `criar_memoria_inicial` do módulo WebAssembly, atribuímos o valor na memória compartilhada.

A memória compartilhada é exportada pelo objeto `exports` dentro da instância e pode ser acessada como `memory`.

> De uma forma mais detalhada, a memória é acessada por instruções binárias `load` e `store`. Na Engenharia da Computação, uma arquitetura `load-store` é uma arquitetura de conjunto de instruções que divide as instruções em duas categorias: acesso à memória (para carregar e armazenar entre memória e registradores) e operações ALU (que ocorrem apenas entre registradores). Essas instruções binárias são acessadas com o deslocamento e o alinhamento. O alinhamento está na representação logarítmica de base 2.

É importante lembrar que o WebAssembly atualmente fornece apenas a possibilidade de trabalhar com endereços de 32 bits. No futuro, o WebAssembly fornecerá também endereços de 64 bits.

```
WebAssembly
.instantiateStreaming(fetch(arquivo))
.then(wasm => {
  const { instance } = wasm;
  const { subtracao, criar_memoria_inicial, memory } = instance.exports;

  // Inicializa a memória pelo WebAssembly
  criar_memoria_inicial();

  // Cria uma visualização da memória usando u8 (0 até 255)
  // Note também o `slice` que seleciona 10 itens da lista
  const arrayMemoria = new Uint8Array(memory.buffer, 0)
    .slice(0, 10);

  // Uint8Array(10) [ 0, 0, 0, 0, 0, 85, 0, 0, 0, 0 ]
  console.log(arrayMemoria);

  // 18
  console.log(subtracao(28, 10));
});
```

Com isso, podemos inicializar a memória pelo lado do WebAssembly e também podemos modificar a memória posteriormente. Que tal salvarmos dados do JavaScript nessa memória? Vamos precisar de duas novas funções (`malloc` e `acumular`) no JavaScript para refletir no nosso arquivo WebAssembly:

```
const {
  subtracao,
  criar_memoria_inicial,
  memory,
  // As novas funções: "malloc" e "acumular"
  malloc,
  acumular
} = instance.exports;
```

Essas duas funções serão escritas posteriormente no lado do Rust, mas vamos manter o foco no JS por enquanto e escrever que tipo de dados nós vamos enviar para o WASM. Podemos começar com algo simples, como uma lista de valores de números.

Para enviar para o WASM, precisamos da matriz tipada (*UInt8Array*), nesse caso nomeada como `jsLista` e com os valores 20, 50 e 80. Vale lembrar que os módulos WebAssembly não têm nenhuma pista sobre o comprimento dos objetos que são criados na memória no lado do JavaScript.

O WebAssembly precisa alocar memória e para isso temos que escrever manualmente a alocação e liberação da memória. Nesta etapa, enviamos o comprimento da matriz tipada e alocamos essa memória usando a função `malloc` , que retorna o ponteiro para a localização da memória.

```
// Cria a matriz tipada com valores em u8 (0 a 255)
const jsLista = Uint8Array.from([20, 50, 80]);

// O comprimento da matriz tipada jsLista: 3
const comprimento = jsLista.length;
```

```
// Retorna o endereço da memória
const wasmListaPonteiro = malloc(comprimento);
```

Em seguida, criamos uma nova matriz tipada com o *buffer* (total de memória disponível), deslocamento de memória (`wasmListaPonteiro`) e o comprimento da memória.

```
// Cria matriz tipada baseada no total da memória disponível, deslocamento
e comprimento
const wasmLista = new Uint8Array(
  memory.buffer,
  wasmListaPonteiro,
  comprimento
);

// Aplica o valor da jsLista no wasmLista
wasmLista.set(jsLista);
```

Estamos enviando o ponteiro e o comprimento para o módulo WebAssembly. No módulo WebAssembly, buscamos o valor da memória e a usamos. Precisamos escrever então nossas novas funções no Rust (`malloc` e `acumular`). Vamos começar com `malloc` , que será importante para salvar os dados da imagem posteriormente:

```
use std::alloc::{alloc, Layout};
use std::mem;

#[no_mangle]
extern fn malloc(comprimento: usize) -> *mut u8 {
    let alinhamento = mem::align_of::<usize>();
    if let Ok(layout) = Layout::from_size_align(comprimento, alinhamento)
{
        unsafe {
            if layout.size() > 0 {
                let ponteiro = alloc(layout);
                if !ponteiro.is_null() {
                    return ponteiro;
                }
```

```
            } else {
                return alinhamento as *mut u8;
            }
        }
    }
    std::process::abort()
}
```

Dado o comprimento da memória, a função `malloc` aloca um bloco de memória e retorna um ponteiro mutável. Desta forma, podemos ler a memória usando o ponteiro e o comprimento. Perceba a função `acumular`, a seguir, que é bem similar à `malloc`. A diferença é que, em vez de realizar a escrita, faz a leitura dos dados.

```
use core::slice::from_raw_parts_mut;

#[no_mangle]
extern fn acumular(ponteiro: *mut u8, comprimento: usize) -> i32 {
    let fatia = unsafe { from_raw_parts_mut(ponteiro as *mut u8,
comprimento) };
    let mut soma = 0;
    for i in 0..comprimento {
        soma = soma + fatia[i];
    }
    soma as i32
}
```

A função `acumular` recebe o *array* compartilhado e o comprimento. Em seguida, recupera os dados da memória compartilhada, percorre os dados e retorna a soma de todos os elementos passados nos dados.

Depois de fazer a compilação do código Rust com `cargo build --release`, adicionaremos a função `acumular` no JavaScript passando o ponteiro e o comprimento, como no exemplo a seguir:

```
// Soma 20 + 50 + 80 e retorna o valor em i32
const somaEntreItensDaLista = acumular(
  wasmListaPonteiro,
```

```
  comprimento
);

// 150
console.log(somaEntreItensDaLista);
```

Com esse código, podemos encerrar parte da escrita na memória do nosso editor. Até então, nosso código do `editor.js` deve ser similar a este:

```
const arquivo =
  './target/wasm32-unknown-unknown/release/editor.wasm';

WebAssembly
.instantiateStreaming(fetch(arquivo))
.then(wasm => {
  const { instance } = wasm;
  const {
    subtracao,
    criar_memoria_inicial,
    memory,
    malloc,
    acumular
  } = instance.exports;

  criar_memoria_inicial();
  const arrayMemoria = new Uint8Array(memory.buffer, 0)
    .slice(0, 10);
  console.log(arrayMemoria); // 85
  console.log(subtracao(28, 10)); // 18

  const jsLista = Uint8Array.from([20, 50, 80]);
  const comprimento = jsLista.length;
  const wasmListaPonteiro = malloc(comprimento);
  const wasmLista = new Uint8Array(
    memory.buffer, wasmListaPonteiro, comprimento
  );

  wasmLista.set(jsLista);
```

```
  const somaEntreItensDaLista = acumular(wasmListaPonteiro, comprimento);
  console.log(somaEntreItensDaLista); // 150
});
```

E o resultado no navegador deve ser igual ao da imagem a seguir:

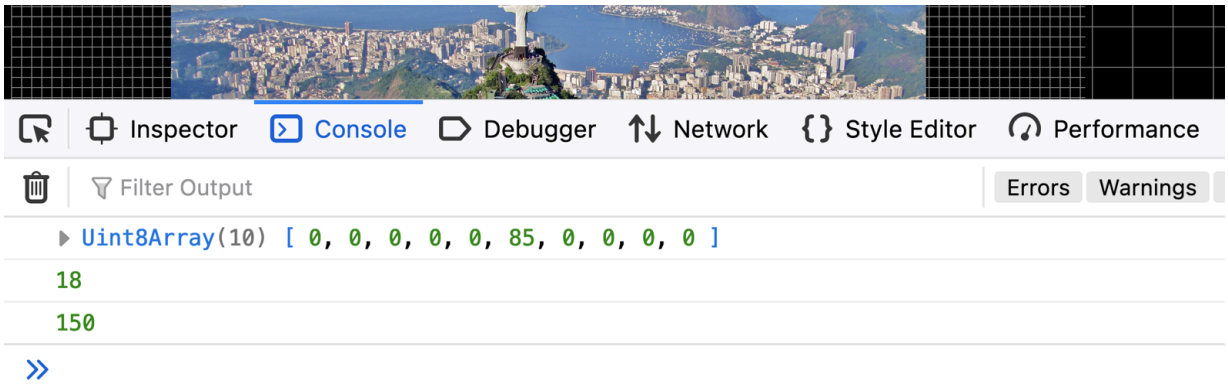


Figura 4.4: Resultado acumulador.

Sensacional! Agora que sabemos que podemos salvar dados do lado do JavaScript e manipular no WebAssembly (pelo uso das funções compiladas do Rust), vamos atualizar nosso código para salvar a imagem que estamos vendo no editor.

4.3 Salvando os dados da imagem

Vamos reutilizar no editor parte da lógica que escrevemos anteriormente, especialmente a função `malloc`, para salvar os dados da imagem.

Daremos vida ao nosso editor criando os seletores e eventos para alterar a imagem quando um novo arquivo de imagem for selecionado. Também adicionaremos a funcionalidade do botão "resetar filtros", que basicamente funciona trocando o atributo `src` da imagem para o último valor da variável `imagemOriginal`, que muda toda vez que um novo arquivo é carregado.

```
const input = document.querySelector('input');

const botaoResetarFiltro = document.querySelector(
  '#remover');
const botaoPBFiltroJs = document.querySelector(
  '#preto-e-branco-js');
const botaoPBFiltroWasm = document.querySelector(
  '#preto-e-branco-wasm');

// Salva o atributo 'src' da imagem original.
let imagemOriginal = document.getElementById('imagem').src;

// Sempre que o valor do botão "input" for alterado,
// o código desta função será executado.
input.addEventListener('change', (event) => {
  const arquivo = event.target.files[0];
  const reader = new FileReader();

  // Seleciona o elemento imagem e atualiza
  // o título baseado no arquivo.
  const imagem = document.getElementById('imagem');
  imagem.title = arquivo.name;

  reader.onload = (event) => {
    // Quando o processo for finalizado salva o resultado no
    // atributo 'src' da imagem.
    // Também atualiza a variável imagemOriginal.
    imagem.src = event.target.result;
    imagemOriginal = event.target.result;
  };

  reader.readAsDataURL(arquivo);
});

// Sempre que o botão "#remover" for clicado,
// esta função será executada
botaoResetarFiltro.addEventListener('click', (event) => {
  const imagem = document.getElementById('imagem');
  imagem.src = imagemOriginal;
```

```
  console.log('Imagem voltou ao original');
});
```

Se você voltar ao navegador, poderá ver que a imagem muda toda vez que selecionamos uma nova imagem. Ao clicar no botão de resetar, não terá nenhuma mudança porque basicamente as imagens são as mesmas. Vamos criar então nosso primeiro filtro 100% JavaScript.

Uma imagem tem como representação de seus dados uma matriz tipada, cuja visualização é baseada em `u8` - ou seja, valores de 0 a 255, números que se conectam ao canal de cores RGB (vermelho, verde e azul).

Os valores fazem referência a cada um dos canais de cores e quanto da cor é preenchida nesse canal. Por exemplo, se o canal `R` for 0, significa zero quantidade de vermelho, se for 255, significa a quantidade máxima de vermelho nesse canal. Esses canais de cores são misturados para formar uma cor. Por exemplo `128, 0, 128` torna-se roxo. Existe também um canal chamado "Alpha" (alfa, em português), que define a transparência da cor. O número 0 significa sem transparência, enquanto 255, transparência total.

Entretanto, para manipular os dados RGB da imagem de maneira prática, a API de imagens é bem limitada quando queremos aplicar qualquer modificação nos dados da imagem. Por isso, precisamos recorrer a uma API do navegador que existe desde 2004, porém pouco popular entre desenvolvedores Web: *HTML5 Canvas*.

Essa API permite *plotar*, ler os dados da imagem ou até mesmo realizar qualquer tipo de alteração de forma muito prática. Como não temos a imagem no `canvas` e, sim, na tag `image` do HTML, precisamos escrever uma função auxiliar que converte o formato da imagem para `canvas`.

A seguir, temos uma função que recebe uma imagem e a desenha em um canvas virtual, ou seja, sua referência existe apenas na memória, não na página.

```
function converteImagemParaCanvas(imagem) {
  // Cria a referência do canvas
  const canvas = document.createElement('canvas');

  // Seleciona o context 2d do canvas
  const contexto = canvas.getContext('2d');

  // Coloca a largura e altura do canvas similar à imagem
  canvas.width = imagem.naturalWidth || imagem.width;
  canvas.height = imagem.naturalHeight || imagem.height;

  // Desenha a imagem no contexto 2d
  contexto.drawImage(imagem, 0, 0);

  // Retorna tanto o canvas como seu contexto
  return { canvas, contexto };
}
```

Essa função nos permite manipular a referência do canvas e seu respectivo contexto, e esses valores são úteis para a nossa função de processamento e execução do filtro preto e branco na imagem, que vamos escrever na seção a seguir.

4.4 Filtro preto e branco em JavaScript

Existem inúmeras fórmulas que performam um filtro preto e branco nos dados de uma imagem. Para este livro, selecionei a fórmula a seguir, pela sua baixa complexidade e por simplificar a leitura do código em Rust e JavaScript:

```
FiltroPeB = R / 3 + G / 3 + B / 3
```

Com base nessa fórmula, vamos criar um filtro preto e branco no JavaScript que retorna um `base64`.

O formato `base64` é um grupo de esquemas de codificação de binário para texto semelhantes que representam dados binários em um formato de "string ASCII", traduzindo-os em uma representação `radix-64`.

O navegador permite trabalhar também com imagens em `base64`, então podemos aproveitar isso ao nosso favor.

```
function filtroPretoBrancoJS(canvas, contexto) {
  // Pega os dados da imagem
  const dadosDaImagem = contexto
    .getImageData(0, 0, canvas.width, canvas.height);

  // Pega os pixels da imagem
  const pixels = dadosDaImagem.data;

  // Performa a mudança em cada pixel da imagem de
  // acordo com a fórmula vista anteriormente
  for (var i = 0, n = pixels.length; i < n; i += 4) {
    const filtro = pixels[i] / 3 + pixels[i+1] / 3 + pixels[i+2] / 3;
    pixels[i] = filtro;
    pixels[i+1] = filtro;
    pixels[i+2] = filtro;
  }

  // Atualiza o canvas com os novos dados
  contexto.putImageData(dadosDaImagem, 0, 0);

  // Retorna um base64 do canvas
  return canvas.toDataURL('image/jpeg');
}
```

Não podemos nos esquecer de adicionar o `EventListener` de *click* no botão de filtro preto e branco do JavaScript:

```
botaoPBFiltroJs.addEventListener('click', (event) => {
  // Seleciona a imagem
  const imagem = document.getElementById('imagem');
```

```
  // Converte a imagem para canvas
  const { canvas, contexto } = converteImagemParaCanvas(imagem);

  // Recebe o base64
  const base64 = filtroPretoBrancoJS(canvas, contexto);

  // Coloca o novo base64 na imagem
  imagem.src = base64;
});
```

Conseguimos! O botão "resetar filtro" e o botão "filtro preto e branco" usando processamento da imagem pelo JavaScript estão funcionando corretamente!

Antes de prosseguirmos com a implementação em WebAssembly, seria interessante metrificar o tempo que leva o processamento da imagem com JavaScript, assim podemos ver a diferença de tempo entre o WASM e JavaScript na prática! Para tornar isso possível, precisamos criar uma função que reporta o tempo da operação. Veja o código da implementação a seguir:

```
function tempoDaOperacao(inicio, fim, nomeDaOperacao) {
  // Seleciona o elemento #performance
  const performance = document.querySelector('#performance');
  // Muda o texto de #performance para o tempo da execução
  perfomance.textContent = `${nomeDaOperacao}: ${fim - inicio} ms.`;
}
```

Após a adição dessa função, podemos atualizar nossa função de filtro (`filtroPretoBrancoJS`) para salvar o tempo da execução antes e após o processamento usando `performance.now()` :

```
// Salva o tempo do início
const inicio = performance.now();

for (var i = 0, n = pixels.length; i < n; i += 4) {
  const filtro = pixels[i] / 3 + pixels[i+1] / 3 + pixels[i+2] / 3;
  pixels[i] = filtro;
```

```
  pixels[i+1] = filtro;
  pixels[i+2] = filtro;
}

// Salva o tempo do fim
const fim = performance.now();

// Reporta o tempo que levou
tempoDaOperacao(inicio, fim, 'JavaScript Preto e Branco');
```

Com o filtro preto e branco escrito em JavaScript pronto, veremos quanto tempo levou para executar a operação. A seguir, temos uma imagem exemplificando o pós-execução:

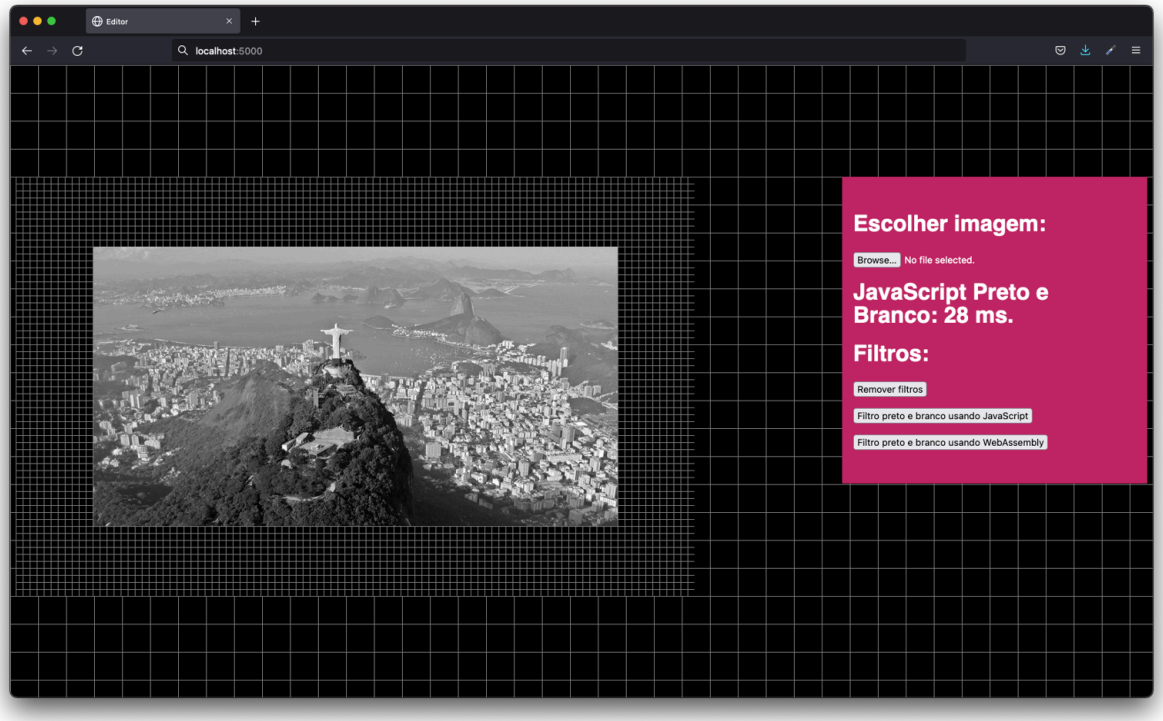


Figura 4.5: Preto e branco JavaScript.

> Nota: se você estiver lendo este livro em preto e branco, é possível ver a imagem em cores no repositório de exemplos do livro.

Note que o tempo muda de acordo com o tamanho da imagem e o poder de processamento que a máquina tem e disponibiliza para a operação realizada pelo navegador, ou seja, em certas máquinas, a operação pode ser mais rápida ou lenta.

Próxima parada: fazer o mesmo filtro, porém escrito em Rust para compilar em formato binário do WebAssembly.

4.5 Filtro preto e branco em WebAssembly

Voltemos para o arquivo `lib.rs` dentro da pasta `src`. O nome da função que vamos escrever é `filtro_preto_e_branco` e será bem similar à nossa versão em JavaScript. Ela também tem a mesma forma de buscar os dados que a função `acumular`, escrita anteriormente, tem.

```
#[no_mangle]
extern fn filtro_preto_e_branco(
    ponteiro: *mut u8, comprimento: usize
) {
    let pixels = unsafe {
        from_raw_parts_mut(ponteiro as *mut u8, comprimento)
    };
    let mut i = 0;
    loop {
        if i >= comprimento - 1 {
            break;
        }
        let filtro = (pixels[i] / 3) + (pixels[i + 1] / 3) + (pixels[i +
2] / 3);
        pixels[i] = filtro;
        pixels[i + 1] = filtro;
        pixels[i + 2] = filtro;
        i += 4;
    }
}
```

Compile novamente o arquivo `src/lib.rs` para o arquivo `editor.wasm` e vamos atualizar o bloco de instanciamento do WebAssembly no `editor.js` para adicionar a nova função criada. Podemos aproveitar e adicionar o *EventListener* dentro do bloco para reusar o escopo de alguns valores:

```
WebAssembly
.instantiateStreaming(fetch(arquivo))
.then(({ instance }) => {
  const {
    subtracao,
    criar_memoria_inicial,
    memory,
    malloc,
    acumular,
    filtro_preto_e_branco
  } = instance.exports;

  botaoPBFiltroWasm.addEventListener('click', (event) => {
    // Seleciona a imagem
    const imagem = document.getElementById('imagem');

    // Converte a imagem para canvas
    const { canvas, contexto } = converteImagemParaCanvas(imagem);

    // Dados da imagem (objeto com tipo ImageData)
    const dadosDaImagem = contexto.getImageData(
        0, 0, canvas.width, canvas.height);

    // Desta vez, usamos "buffer" em vez de "data" para
    // poder montar a matriz tipada da imagem
    const buffer = dadosDaImagem.data.buffer;

    // Matriz tipada de u8 (0 a 255)
    const u8Array = new Uint8Array(buffer);

    // Realiza o malloc matriz tipada
    const ponteiro = malloc(u8Array.length);
```

```
    // Cria matriz tipada a partir da visualização da memória
    // da instância, ponteiro e comprimento (u8Array.length)
    let wasmArray = new Uint8ClampedArray(
        instance.exports.memory.buffer,
        ponteiro,
        u8Array.length
    );

    // O valor de wasmArray se torna o mesmo da nossa imagem
    wasmArray.set(u8Array);
  });

  // ... código anterior ...
```

Há uma novidade nesse código: a adição do uso do `Uint8ClampedArray`. Ele é bem similar ao `Uint8Array`, porém, se você está tentando definir um elemento para uma matriz tipada fixada de `Uint8` (`Uint8ClampedArray`) com qualquer valor fora do intervalo de 0 a 255, ele simplesmente assumirá como padrão 0 ou 255 (dependendo de o valor ser menor ou maior). Já uma matriz `Uint8Array` apenas recebe os primeiros 8 bits do valor.

Para entender melhor, vamos a um rápido exemplo da diferença descrita entre o `Uint8ClampedArray` e `Uint8Array`:

```
const x = new Uint8ClampedArray([17, -45.3]);
console.log(x[0]); // 17
console.log(x[1]); // 0
console.log(x.length); // 2

const y = new Uint8Array([17, -45.3]);
console.log(y[0]); // 17
console.log(y[1]); // 211
console.log(y.length); // 2
```

Por que precisamos do `Uint8ClampedArray`? Para criar dados da imagem de forma correta usando o tipo `ImageData` do JavaScript, é necessário usar `Uint8ClampedArray` em vez de `Uint8Array`. Usando a matriz tipada fixada, podemos plotar no canvas imagens criadas no

navegador. Vamos terminar de escrever a função do filtro dentro de *EventListener* de *click*:

```
// ...

// O valor de wasmArray se torna o mesmo da nossa imagem
wasmArray.set(u8Array);

// Salva o tempo do início da operação
const inicio = performance.now();

// Aplica filtro na imagem que está salva na memória
filtro_preto_e_branco(ponteiro, u8Array.length);

// Salva o tempo do final da operação
const final = performance.now();

// Computa tempo da operação
tempoDaOperacao(inicio, final, 'WebAssembly Preto e Branco');
```

Até então, mapeamos a memória no WebAssembly para refletir o mesmo valor que a visualização dos dados da imagem. Após isso, executamos a função que modifica a memória da imagem e aplica o filtro, nossa função `filtro_preto_e_branco`.

Falta apenas criar uma nova imagem baseada no novo valor da matriz tipada fixa:

```
// ...

// Computa tempo da operação
tempoDaOperacao(inicio, final, 'WebAssembly Preto e Branco');

// Pega altura e largura da imagem
const width = imagem.naturalWidth || imagem.width;
const height = imagem.naturalHeight || imagem.height;

// Cria dados da imagem (ImageData) usando largura e altura
const novoDadosDaImagem = contexto.createImageData(width, height);
```

```
// Preenche os dados com a matriz tipada
novoDadosDaImagem.data.set(wasmArray);

// Atualiza o canvas com a nova imagem
contexto.putImageData(novoDadosDaImagem, 0, 0);

// Atualiza o atributo src como o base64 do canvas
imagem.src = canvas.toDataURL('image/jpeg');
```

A peça que faltava para o nosso filtro era a adição do bloco de código que acabamos de escrever. Finalmente! Que tal vermos a diferença entre ambos os filtros no navegador?

O filtro preto e branco está funcionando corretamente com WebAssembly e seu tempo de execução é mais rápido do que com o JavaScript! Bem, isso é algo que já esperávamos, dado que o WASM foi criado para esse tipo de operação.

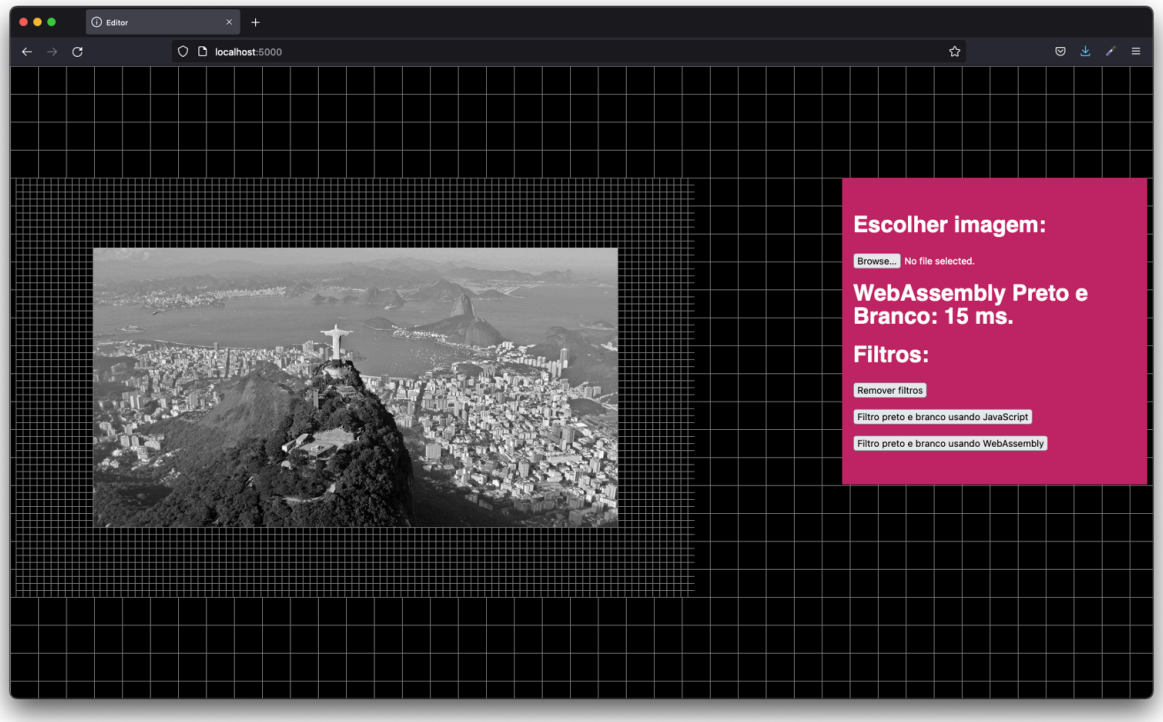


Figura 4.6: Preto e branco WebAssembly.

Nos casos gerais, o WebAssembly é mais performático, porém o tempo de execução do WASM pode até ser similar ao do JavaScript, dependendo do poder de processamento disponibilizado pela máquina para uso no navegador.

4.6 Filtros de tons de cores em WebAssembly

Para aplicar o filtro preto e branco, estamos reduzindo os valores dos canais de cores do RGBA (com exceção do valor de alfa) para deixar o mais próximo possível de 0 (preto); considerando essa informação, podemos assumir que é possível destacar tons na imagem baseado em qualquer cor que quisermos.

Escreveremos nesta seção três filtros que relevam tons de cores diferentes: vermelho, verde e azul. Iniciaremos escrevendo a lógica destes filtros no lado do Rust, começando com a função de processamento para destacar o tom vermelho:

```
#[no_mangle]
extern fn filtro_vermelho(data: *mut u8, len: usize) {
    let pixels = unsafe {
        from_raw_parts_mut(data as *mut u8, len)
    };
    let mut i = 0;
    loop {
        if i >= len - 1 {
            break;
        }

        // 0
        pixels[i + 1] = pixels[i + 1] / 2;
        pixels[i + 2] = pixels[i + 2] / 2;
        // 3
        // 4
        pixels[i + 5] = pixels[i + 5] / 2;
        pixels[i + 6] = pixels[i + 6] / 2;
```

```
        // 7
        i += 8;
    }
}
```

Mudamos algumas coisas nessa função em comparação com a função preto e branco que escrevemos anteriormente: a forma como percorremos o laço e a modificação apenas nos canais verde e azul (segundo e terceiro canal, respectivamente). A redução dos demais canais de cores, com exceção do canal vermelho, dará relevância ao tom vermelho na imagem.

Os demais filtros deverão seguir o mesmo padrão: se estamos enfatizando a cor verde, logo todos os outros canais de cores deverão ser reduzidos. A seguir, veja o código dos filtros de destaque dos tons verde e azul:

```
#[no_mangle]
extern fn filtro_verde(data: *mut u8, len: usize) {
    let pixels = unsafe {
        from_raw_parts_mut(data as *mut u8, len)
    };
    let mut i = 0;
    loop {
        if i >= len - 1 {
            break;
        }

        pixels[i] = pixels[i] / 2;
        // 1
        pixels[i + 2] = pixels[i + 2] / 2;
        // 3
        pixels[i + 4] = pixels[i + 4] / 2;
        // 5
        pixels[i + 6] = pixels[i + 6] / 2;
        // 7
        i += 8;
    }
}
```

```
#[no_mangle]
extern fn filtro_azul(data: *mut u8, len: usize) {
    let pixels = unsafe {
        from_raw_parts_mut(data as *mut u8, len)
    };
    let mut i = 0;
    loop {
        if i >= len - 1 {
            break;
        }

        pixels[i] = pixels[i] / 2;
        pixels[i + 1] = pixels[i + 1] / 2;
        // 2
        // 3
        pixels[i + 4] = pixels[i + 4] / 2;
        pixels[i + 5] = pixels[i + 5] / 2;
        // 6
        // 7
        i += 8;
    }
}
```

Após a adição dessas duas funções, vamos mais uma vez gerar nosso arquivo binário WebAssembly com o comando de compilação e atualizar as funções de filtro que acabamos de criar no JavaScript.

A primeira coisa que precisamos fazer é dar uma reorganizada no código JavaScript com foco em não escrever longas funções para cada vez que adicionarmos um filtro.

No código a seguir, escrevemos diversas chamadas para a função `adicionarFiltro` (vamos escrever sua respectiva lógica depois), que recebe os respectivos parâmetros: a descrição do filtro, a função do filtro e o elemento de botão que aciona o filtro.

```
WebAssembly
.instantiateStreaming(fetch(arquivo))
```

```
.then(({instance}) => {
  const {
    // ... outros exports
    // Adicione os três novos filtros
    filtro_vermelho,
    filtro_verde,
    filtro_azul
  } = instance.exports;

  adicionarFiltro('Preto e Branco WASM', '#preto-e-branco-wasm', {
    instance, filtro: filtro_preto_e_branco
  });
  adicionarFiltro('Vermelho WASM', '#vermelho-wasm', {
    instance, filtro: filtro_vermelho
  });
  adicionarFiltro('Azul WASM', '#azul-wasm', {
    instance, filtro: filtro_azul
  });
  adicionarFiltro('Verde WASM', '#verde-wasm', {
    instance, filtro: filtro_verde
  });
```

Com esse código, estamos adicionando o filtro com a invocação da função `adicionarFiltro` em vez de escrever o bloco de inicialização do filtro para cada filtro adicionado. A função `adicionarFiltro` terá o mesmo bloco de inicialização que estávamos fazendo para um botão apenas (`#preto-e-branco-wasm`), porém ela configura o filtro com base nos parâmetros enviados à função.

```
function executarFiltro(image, processImageFn) {
  const { canvas } = converteImagemParaCanvas(image);
  if (!processImageFn) {
    return canvas.toDataURL();
  }

  if (typeof processImageFn === 'function') {
    processImageFn(canvas, canvas.getContext('2d'));
    return canvas.toDataURL('image/jpeg');
  }
```

```
}

function adicionarFiltro(text, selector, { instance, filtro }) {
  const button = document.querySelector(selector);
  const imagem = document.getElementById('imagem');
  button.addEventListener('click', () => {
    executarFiltro(imagem, (canvas, context) => {
    const image = document.getElementById("imagem");
    const imageData = context.getImageData(0, 0, canvas.width,
canvas.height);
    const buffer = imageData.data.buffer;
    const u8Array = new Uint8Array(buffer);
    let wasmClampedPtr = instance.exports.malloc(u8Array.length);
    let wasmClampedArray = new
Uint8ClampedArray(instance.exports.memory.buffer, wasmClampedPtr,
u8Array.length);
    wasmClampedArray.set(u8Array);
    const startTime = performance.now();
    filtro(wasmClampedPtr, u8Array.length);
    const endTime = performance.now();
    tempoDaOperacao(startTime, endTime, text);
    const width = image.naturalWidth || image.width;
    const height = image.naturalHeight || image.height;
    const newImageData = context.createImageData(width, height);
    newImageData.data.set(wasmClampedArray);
    context.putImageData(newImageData, 0, 0);
    image.src = canvas.toDataURL('image/jpeg');
    });
  });
}
```

Agora que temos uma nova forma de associar filtros do WebAssembly aos eventos de *click* em botões, lembre-se de deletar a forma como adicionamos o evento da função filtro preto e branco do WASM diretamente no botão `botaoPBFiltroWasm`. Faça a deleção deste bloco de código:

```
botaoPBFiltroWasm.addEventListener('click', (event) => {
  const imagem = document.getElementById('imagem');
```

```
  const { canvas, contexto } = converteImagemParaCanvas(imagem);
  const dadosDaImagem = contexto.getImageData(
    0, 0, canvas.width, canvas.height);
  const buffer = dadosDaImagem.data.buffer;
  const u8Array = new Uint8Array(buffer);
  const ponteiro = malloc(u8Array.length);
  let wasmArray = new Uint8ClampedArray(
    instance.exports.memory.buffer,
    ponteiro,
    u8Array.length
  );
  wasmArray.set(u8Array);
});
```

> Caso precise, consulte o código no repositório de exemplos do livro no Github: https://github.com/raphamorim/livro-webassembly-exemplos/.

Para fechar, precisamos adicionar os botões dos novos filtros no `index.html`:

```
<h1>Filtros:</h1>
<p><button id="remover">Remover filtros</button></p>
<p><button id="preto-e-branco-js">Filtro preto e branco usando
JavaScript</button></p>
<p><button id="preto-e-branco-wasm">Filtro preto e branco usando
WebAssembly</button></p>
<p><button id="vermelho-wasm">Filtro vermelho usando WebAssembly</button>
</p>
<p><button id="verde-wasm">Filtro verde usando WebAssembly</button></p>
<p><button id="azul-wasm">Filtro azul usando WebAssembly</button></p>
```

Após todas essas etapas, já podemos testar as mudanças no navegador. Os novos botões deverão funcionar da forma esperada. Veja uma imagem do resultado após o processamento do filtro de realce de tom vermelho.

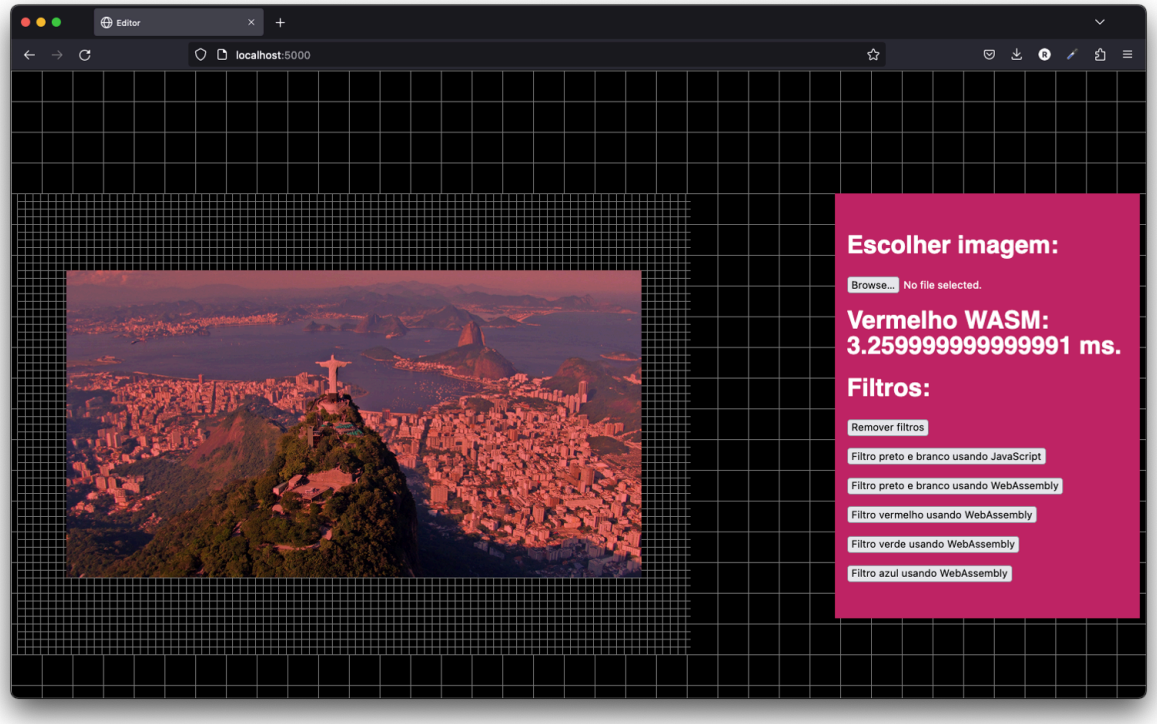


Figura 4.7: Filtro vermelho em WebAssembly

> Nota: é bom relembrar que, se você estiver lendo o livro em preto e branco, é possível ver a imagem em cores no repositório de exemplos do livro.

Até então, todos os filtros que escrevemos aplicaram mudanças nos canais de cores vermelho, verde ou azul, mas nunca no canal alfa. O canal alfa é um componente de cor que representa o grau de transparência (ou opacidade) de uma cor. O próximo filtro que vamos escrever, o *filtro de opacidade*, aplica mudanças nele.

4.7 Filtro de opacidade em WebAssembly

Como vimos anteriormente, as cores são representadas em formato digital como uma coleção de números, cada valor representa o nível de intensidade de um determinado canal de cor. Nos filtros de tons de cores da seção anterior, pudemos ver a diferença na expressividade de uma cor quando reduzimos o valor do canal.

A cor preta é composta com o menor valor possível nos canais de cores (vermelho, verde e azul). Podemos representar o valor decimal do RGB da cor preta em `(0,0,0)`, assim como o inverso do preto, o branco, é composto pelo valor máximo dos canais de cores: `(255,255,255)`.

Os valores de cores **RGBA** são uma extensão dos valores de cores RGB com um canal alfa, que especifica a opacidade dos canais de cores. A imagem a seguir exemplifica diferentes valores do RGBA seguindo com a cor preta `(0,0,0)`:

Figura 4.8: RGBA cor preta.

As cores mudam drasticamente com a aplicação de diferentes valores do canal alfa. Na figura, vimos os valores RGB baseados no espaço de cor conhecido como *sRGB*, formato implementado pela HP e Microsoft em 1996 que eventualmente se tornou parte da especificação da Web. Na Web, os possíveis valores do canal alfa são diferentes dos canais de cores (0 a 255) e podem ser valores de 0 a 1.

> Curiosidade: o canal alfa é muito usado na computação gráfica na competência conhecida como composição alfa. A composição alfa combina uma imagem com um plano de fundo para criar a aparência de transparência parcial ou total. É útil para renderizar pixels de diferentes imagens em uma ou mais camadas e, em seguida, combinar as imagens em uma única imagem final chamada composição — amplamente usada em filmes ao combinar elementos de imagem renderizados por computador com filmagens ao vivo.

Todos os filtros que escrevemos seguem o mesmo padrão de valores da Web. No navegador, é possível ver como o canal alfa funciona usando a função `rgba()` no CSS. Os possíveis valores do canal alfa no CSS estão entre 0 e 1. A imagem demonstra o formato RGBA do valor `(0,0,0,0.8)` no navegador:

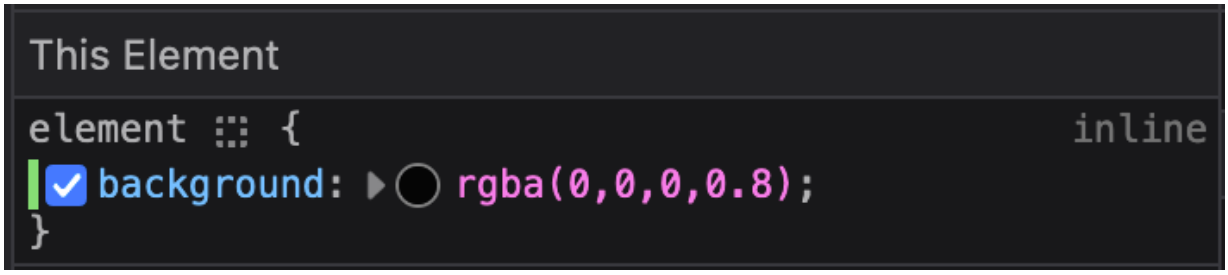


Figura 4.9: RGBA na Web.

Nosso editor de imagem segue o padrão da Web, que se baseia em canais de cores com valores `u8` ; porém, se quiséssemos endereçar o processamento de imagem para além da Web, poderíamos usar

outros espaços de cores, como o espaço de cor da Adobe (implementado em 1998), que provê mais cores que o sRGB; entretanto, há problemas na fidelidade de cores quando o formato não é suportado. O espaço de cores da Adobe é um dos formatos preferidos por profissionais que trabalham frequentemente com processamento de imagens.

Se a mudança no canal alfa seguir o mesmo padrão da função do CSS, teremos um problema diretamente com a arquitetura atual do nosso editor de imagem. Considerando que valores alfa no CSS podem ser entre 0 a 1, é certo assumir que esses valores são representados por números flutuantes, e nossas visualizações de memória foram criadas baseadas no uso de dados em `u8`.

Para nossa sorte, esse problema é automaticamente resolvido pelo navegador assim que usamos `Uint8ClampedArray` para criar os dados da imagem. Os valores do canal alfa são de 0 a 255, em vez do padrão do CSS, que usa números flutuantes. Por ora, esse problema foi resolvido de forma automática, mas existem casos nos quais os dados de uma imagem não são originalmente necessariamente `u8`, ou nos quais o processamento segue uma fórmula baseada em números flutuantes (problema comum com a versão popular da fórmula de filtro sépia), desencadeando problemas de tipos de valores.

Sabemos, então, que podemos usar valores de 0 a 255 no canal alfa assim como nos outros canais de cores, logo a estrutura da nossa função de filtro de opacidade deve ser parecida com as demais funções que escrevemos anteriormente. No código a seguir, temos nossa função de processamento de imagem nomeada como `filtro_opacidade` e com a finalidade de reduzir o valor do canal alfa de 10 em 10 sempre que possível:

```
#[no_mangle]
extern fn filtro_opacidade(data: *mut u8, len: usize) {
    let pixels = unsafe {
        from_raw_parts_mut(data as *mut u8, len)
```

```
    };
    let mut i = 0;
    let alfa = 10;
    loop {
        if i >= len - 1 {
            break;
        }

        let valor_atual = pixels[i + 3];
        if valor_atual >= alfa {
            pixels[i + 3] = valor_atual - alfa;
        } else {
            pixels[i + 3] = 0;
        };

        i += 4;
    }
}
```

Esse código contém uma validação para ver se o canal alfa atual é maior do que a redução, pois não queremos ter um caso onde o canal alfa seja menor que zero, levando a um erro em tempo de execução. Compile o arquivo `src/lib.rs` com o `cargo build --release` e vamos atualizar o arquivo JavaScript e `index.html`. No arquivo JavaScript, o novo filtro será adicionado da mesma forma como adicionamos os filtros de cores:

```
WebAssembly
.instantiateStreaming(fetch(arquivo))
.then(({instance}) => {
  const {
    // ... outros exports
    // Adicione o filtro de opacidade
    filtro_opacidade
  } = instance.exports;

adicionarFiltro('Opacidade WASM', '#opacidade-wasm', {
  instance, filtro: filtro_opacidade
});
```

Dentro do `index.html`, basta atualizar a lista de botões com o botão de opacidade dentro do elemento da classe *painel*:

```
    <!-- ... -->
    <p><button id="super-azul-wasm">Filtro super azul usando
WebAssembly</button></p>
    <p><button id="opacidade-wasm">Filtro opacidade usando
WebAssembly</button></p>
</div>
```

Salve os arquivos e atualize a página do navegador. Vamos dar uma olhadinha em como o filtro de opacidade funciona. Igual aos demais filtros (com exceção do "super azul"), estamos sempre aplicando iterações de valores em canais para cada vez que o botão é clicado; isso significa que você pode clicar diversas vezes no botão do filtro de opacidade para ver diferentes mudanças serem aplicadas na imagem.

Até então, temos o filtro preto e branco, os filtros de tons de cores e o filtro de opacidade. Estamos quase no fim do capítulo e mais dois filtros serão adicionados, atualizaremos o estilo do elemento com a classe *painel* para permitir ter mais botões sem precisar aumentar o tamanho da página para cada novo botão adicionado.

No nosso arquivo CSS, vamos atualizar a classe *painel*. Adicione um tamanho fixo de *450 pixels*, juntamente da propriedade `overflow-y` com o valor `scroll`. A propriedade CSS `overflow-y` define o que fazer quando o conteúdo transborda as bordas superior e inferior de um elemento em nível de bloco. As possíveis opções são: fazer nada, adicionar uma barra de rolagem ou permitir que o conteúdo exceda a borda. No nosso caso, queremos habilitar o *scroll*, como mostra o código a seguir:

```
.painel {
    padding: 2em 1em;
    background: #c12664;
    float: right;
    width: 25%;
```

```
    /* Novos estilos */
    height: 450px;
    overflow-y: scroll;
}
```

Bem legal! Os novos estilos que incrementamos no editor nos permitem continuar adicionando filtros sem nos preocuparmos com um painel excedente ao tamanho inicial da página. Na próxima seção, vamos escrever o último filtro do editor de imagens: o filtro de inversão.

4.8 Filtro de inversão em WebAssembly

O filtro de inversão inverte os valores dos canais de cores de uma imagem. Alguns exemplos de inversão são:

- Se o valor do canal é 0, torna-se 255;
- Se o valor do canal é 255, torna-se 0;
- Se o valor do canal é 85, torna-se 170.

O valor da inversão é baseado na subtração do número máximo possível de um canal com o valor do canal. O código a seguir mostra a implementação do filtro de inversão no Rust:

```
#[no_mangle]
extern fn filtro_inversao(data: *mut u8, len: usize) {
    let pixels = unsafe {
        from_raw_parts_mut(data as *mut u8, len)
    };
    for i in (0..len - 1).step_by(4) {
        pixels[i] = 255 - pixels[i];
        pixels[i + 1] = 255 - pixels[i + 1];
        pixels[i + 2] = 255 - pixels[i + 2];
```

```
    }
}
```

Compilaremos o projeto pela última vez e atualizaremos nosso arquivo JavaScript e HTML com a nova função de filtro adicionada, além do botão que aciona o filtro no `index.html` :

index.html

```
    <!-- ... -->
    <p><button id="inversao-wasm">Filtro inversão usando
WebAssembly</button></p>
</div>
```

editor.js

```
WebAssembly
.instantiateStreaming(fetch(arquivo))
.then(({instance}) => {
  const {
    // ... outros exports
    // Adicione o filtro inversão
    filtro_inversao
  } = instance.exports;

adicionarFiltro('Inversão WASM', '#inversao-wasm', {
  instance, filtro: filtro_inversao
});
```

Atualize o navegador e vamos testar a função de inversão clicando no botão do filtro.

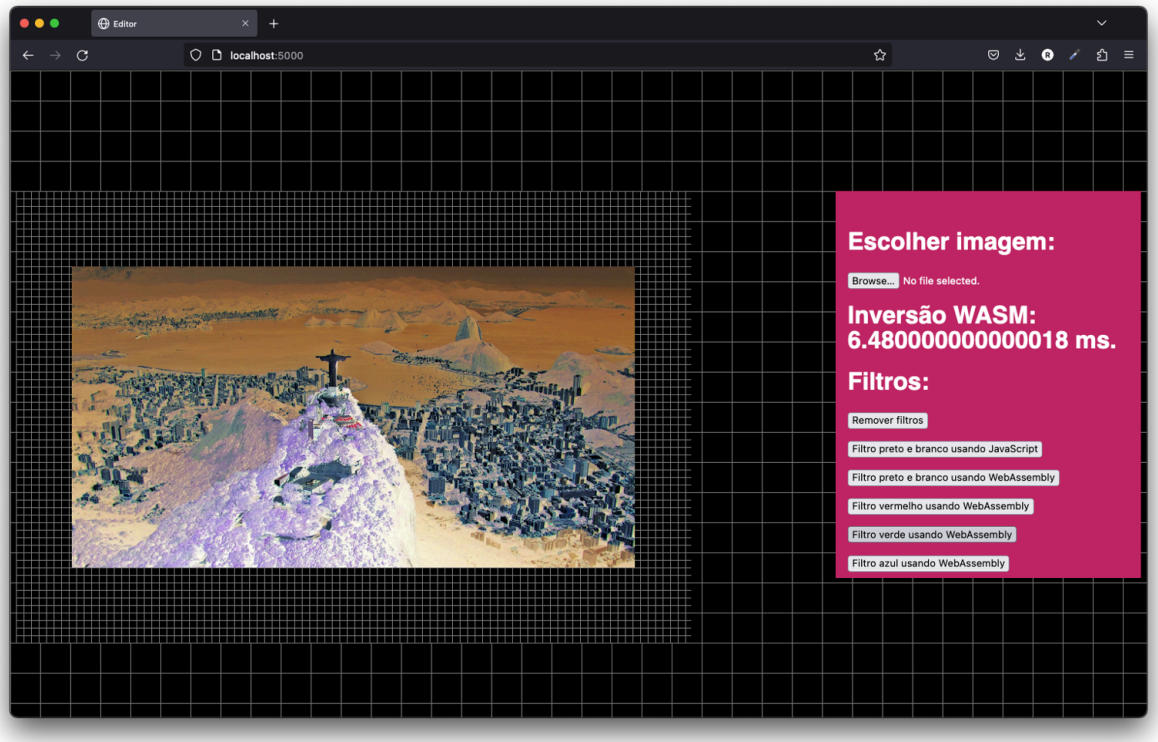


Figura 4.10: Exemplo filtro de inversão WebAssembly.

> Lembrete: é possível ver essa imagem em cores no repositório de exemplos do livro.

Se o seu resultado é similar ao dessa imagem, podemos comemorar, pois finalmente completamos o nosso editor de filtros de imagem em WebAssembly, juntamente com alguns filtros escritos em JavaScript.

Neste capítulo, pudemos aprender como o WebAssembly conversa com o JavaScript no navegador de forma prática, realizando operações de processamento fora do JS e passando dados do JavaScript para o WASM e vice-versa.

Há inúmeras opções que não exploramos neste capítulo e que poderiam otimizar o editor de imagens, como a exploração em bibliotecas nativas com vinculação, otimização do arquivo binários,

tratamento de erro, possibilidade de processamento paralelo ou mesmo manipulação de valores em tipos além do `u8`.

Vamos ver cada uma dessas competências no decorrer do livro, com exemplos que enriquecem o conhecimento de como trabalhar com WebAssembly em diferentes áreas além da manipulação de imagem, mas recomendo, assim que terminar a leitura do livro, revisitar o editor de imagens e colocar todo o conhecimento obtido em prática para aplicar melhorias.

No próximo capítulo, vamos nos aprofundar na comunicação entre bibliotecas dinâmicas e estáticas usando o conceito de vinculação para criar diferentes possibilidades no dia a dia do desenvolvimento.

Capítulo 5
Vinculação e interface binária de aplicativo

Neste capítulo, vamos ver como funcionam os conceitos de *vinculação* e *interface binária* de aplicativo no Rust e como usá-los quando pensamos e desenvolvemos aplicações com WebAssembly. Os dois conceitos são importantíssimos e amplamente utilizados.

Na Ciência da Computação, **vinculação** é a determinação de uma referência ou associação entre uma entidade de um programa (variáveis, rotinas e sentenças) e seus atributos. Existem dois tipos de vinculação: *dinâmica* e *estática*.

A **vinculação estática** ocorre em tempo de compilação e a **vinculação dinâmica** ocorre em tempo de execução. Por isso, elas também são chamadas de ligação inicial e tardia, respectivamente. Para a vinculação estática acontecer, todas as informações necessárias para chamar uma função precisam estar disponíveis em tempo de compilação. Já a vinculação dinâmica, como ela ocorre no tempo de execução, acaba sendo usada quando o compilador não pode determinar todas as informações necessárias para uma chamada de função no tempo de compilação.

> Por que é importante entender a vinculação de bibliotecas? O que isso tem a ver com WebAssembly?
>
> Aprender o conceito de vinculação permite portar código já existente para Web, expandindo o número de possibilidades de aplicações no navegador.

O compilador do Rust suporta tanto vinculação dinâmica como estática, podendo gerar vários artefatos pela linha de comando (por exemplo, usando o parâmetro `--crate-type=cdylib`) ou escrevendo atributo `#[crate_type = "cdylib"]` no código Rust.

Até então, neste livro usamos exclusivamente `cdylib`, porém existem outros `crate_type` disponíveis como opção no compilador. Veja as opções de saída de artefato que o `rustc` provê:

- `bin`: vincula todas as dependências Rust e nativas, produzindo um único binário distribuível;
- `lib`: produz uma biblioteca Rust que é baseada no estilo de biblioteca "recomendada pelo compilador", ou seja, sempre poderá ser usada pelo `rustc`. No entanto, o tipo real da biblioteca pode mudar ao longo do tempo, o que significa que o tipo de biblioteca é definido pelo compilador (podendo ser qualquer um dos outros tipos de biblioteca);
- `dylib`: produz uma biblioteca Rust dinâmica, mas é diferente do `lib`, pois declara o tipo da vinculação exclusivamente dinâmica. Esse tipo de saída cria arquivos com a extensão `.so` no Linux, arquivos com a extensão `.dylib` no macOS e arquivos com extensão `.dll` no Windows;
- `cdylib`: produz uma biblioteca de sistema dinâmica. Os formatos de arquivos de saída do `cdylib` são basicamente os mesmos do `dylib`. A diferença principal é que o `cdylib` foi projetado para criar uma biblioteca dinâmica que pode ser vinculada em outras linguagens como C ou C++;
- `staticlib`: produz uma biblioteca de sistema estática, diferente das outras saídas de biblioteca, pois o compilador nunca tentará vincular a saídas `staticlib`. Cria formatos de arquivos `.a` no Linux, macOS e Windows (MinGW) e também formatos de arquivos `.lib` no Windows (MSVC, compilador C e C++ da Microsoft);
- `rlib`: produz um arquivo com um comportamento conhecido como *Rust Library*, que é um arquivo intermediário e funciona como se fosse uma biblioteca Rust estática. Assim como o `staticlib`, é usado para produzir executáveis vinculados

estaticamente. A única diferença é que o arquivo Rust Library é interpretado apenas pelo compilador Rust;
- `proc-macro` : dependendo dos parâmetros usados para compilação, produz uma macro procedural. Basicamente, é uma maneira de estender o compilador Rust e fornecer plug-ins.

Você não precisará da maioria dos destinos de saída apresentados aqui quando for trabalhar com WebAssembly usando Rust. Até então usamos apenas `cdylib` .

Ok, mas como a vinculação se conecta com a interface binária de aplicativo?

Uma interface binária de aplicativo (ou ABI, do inglês, *Application Binary Interface*) descreve a interface de baixo nível entre uma aplicação e o sistema operacional, entre a aplicação e suas bibliotecas ou entre componentes de uma aplicação.

Ou seja, uma ABI define os mecanismos pelos quais as funções são invocadas e como os parâmetros são passados. A ABI também controla como os programas são carregados na memória e como as bibliotecas são implementadas — em outras palavras, define o que será vinculado ou não.

No Rust, você pode especificar a ABI em funções externas usando a palavra-chave `extern` . O código a seguir fornece um exemplo de duas configurações de ABI:

```
// Declara uma função com a ABI "C"
extern "C" fn novo_i32() -> i32 { 0 }

// Declara uma função com a ABI "system"
extern "system" fn novo_i32_system() -> i32 { 0 }

// Declara uma função com a ABI "stdcall" (Windows API)
extern "stdcall" fn novo_i32_stdcall() -> i32 { 0 }
```

Esses exemplos de código demonstram como as funções podem ser exportadas para o uso de outras linguagens de programação, como C, pois usam uma ABI diferente da que o Rust permite.

Existem três configurações de ABI que são multiplataforma e às quais todos os compiladores têm garantia de suporte:

- `Rust` : a ABI padrão quando você escreve um `fn foo()` normal em qualquer código Rust;
- `C` : é o mesmo que `extern fn foo();` e segue qualquer que seja o padrão que seu compilador C suporta;
- `system` : geralmente funciona de forma similar ao extern `C` , exceto no *Win32*, caso em que é `stdcall` .

No passado, foi muito comum (especificamente nas primeiras versões do destino de compilação `wasm32-unknown-unknown`) o Rust usar a ABI configurada como `extern "C"` , pela maior portabilidade. Outro grande fator era a possibilidade de vincular com funções do *Emscripten*, compilador de C e C++ para WebAssembly.

Também existem outras ABIs para plataformas específicas. Veja alguns exemplos:

- `cdecl` : configuração de interface para o código *x86_32 C*;
- `stdcall` : configuração de interface para a API *Win32* em *x86_32*;
- `win64` : configuração de interface para código C no Windows *x86_64*;
- `sysv64` : configuração de interface para código C que não seja Windows *x86_64*;
- `aapcs` : configuração de interface para o ARM.

É importante destacar que qualquer função que faz omissão da definição da ABI está usando de forma implícita o Rust ABI. Isso porque, quando escrevemos uma função simples sem a definição de `extern` , é basicamente declarar a ABI como "Rust", como no exemplo a seguir:

```rust
// Quando escrevemos qualquer função com omissão de "extern"
fn minha_funcao() {}

// é o equivalente a usar "Rust" como ABI
extern "Rust" fn minha_funcao() {}
```

No código do editor de imagens, usamos exclusivamente a palavra-chave `extern` sem especificar de forma declarativa a ABI utilizada. Apesar de uma função sem uso de `extern` usar Rust ABI, não é o mesmo comportamento quando a função usa `extern`.

A função que tem a palavra-chave `extern` está apontando para a *C ABI*; logo, nosso editor de imagens foi escrito usando a *C ABI* em vez do *Rust ABI*. Provavelmente você deve estar se perguntando: por quê? Para responder essa pergunta, vamos ver a diferença das duas ABIs no contexto do WebAssembly.

5.1 Uso da C ABI e Rust ABI no WebAssembly

A *C ABI* é uma escolha de longa data para chamadas de funções interlinguagens. A *C ABI* não mudará, então podemos ter certeza de que nossa interface do módulo WebAssembly também não mudará. Se escrevermos uma função de soma de `u8` para usar a *C ABI* declarativamente, ela deverá ser igual a este código:

```rust
#[no_mangle]
pub extern "C" fn soma(numero_a: u8, numero_b: u8) -> u8 {
    numero_a + numero_b
}
```

Assim como vimos anteriormente, podemos também omitir o `"C"` e usar apenas usar `extern`, já que *C ABI* é a ABI opção padrão da linguagem, como mostra o exemplo a seguir:

```rust
#[no_mangle]
pub extern fn soma(numero_a: u8, numero_b: u8) -> u8 {
```

```
    numero_a + numero_b
}
```

Nesse caso, usar a ABI do Rust ou do C não faz muita diferença, pois se trata de uma função que retorna um tipo primitivo (`u8`), porém compilar e usar tipos não primitivos do Rust em um binário WebAssembly pode ser uma dor de cabeça se você não tiver o entendimento de como o compilador vai tratar os dados.

Uma recomendação é sempre manter os tipos de valor alinhados com os tipos de dados suportados pelo WebAssembly. Obviamente, o compilador permite que você use tipos de nível superior como parâmetros de função e valores de retorno. O que o compilador vai emitir nesses casos é baseado na ABI que você estiver usando.

Na C ABI, o compilador vai transformar o tipo utilizado em uma referência de ponteiro, por exemplo, os tipos com a característica `Sized` (*structs*, *enums* e outros). Os tipos de *arrays* e tuplas, que também são `Sized` , recebem tratamento especial e são convertidos para um valor imediato se usarem menos de 32 bits.

Entretanto, se você retornar um tipo de *array* maior que 32 bits, a função não receberá nenhum valor de retorno e, em vez disso, obterá um parâmetro de função adicional do tipo `i32` , que será usada como um ponteiro. Se uma função retornar uma tupla, ela sempre se transformará em um parâmetro de função, independentemente do tamanho da tupla.

A seguir, temos três exemplos de alguns dos casos aqui descritos:

```
#[no_mangle]
extern fn ponteiro_usando_box() -> *const [u8; 4] {
    Box::into_raw(Box::new([8,5,8,5]))
}

#[no_mangle]
extern fn ponteiro() -> *const u8 {
    [8,5,8,5].as_ptr()
}
```

```
#[no_mangle]
fn sem_ponteiro() -> [u8; 4] {
    [8,5,8,5]
}
```

Nas primeiras duas funções, estamos retornando ponteiros de forma direta para o JavaScript considerando que os valores serão computados na memória do WebAssembly após a compilação; assim, é possível obter os valores pela referência do ponteiro.

O último caso é uma função sem o uso da palavra-chave `extern` que retorna o valor de forma explícita. De certa forma podemos considerar que são três formas diferentes de retorno, apesar de os códigos da primeira e segunda funções serem similares.

Se fizéssemos o instanciamento dessas funções no JavaScript para tratar as funções que retornam um ponteiro sem `Box`, ponteiro com `Box` e o *Array*, obteríamos resultados bem interessantes. No código a seguir, temos um exemplo de uso para cada função:

```
const {
    ponteiro_usando_box,
    ponteiro,
    sem_ponteiro
} = instance.exports;

// ----
// Ponteiro com Box
const ptrBox = ponteiro_usando_box();
const ptrBoxValor = new Uint8Array(
  instance.exports.memory.buffer, ptrBox, 4);
// ptrBoxValor Uint8Array(4) [ 8, 5, 8, 5 ]
console.log('ptrBoxValor', ptrBoxValor);

// ----
// Ponteiro sem Box
const ptr = ponteiro();
const ptrValor = new Uint8Array(
```

```
  instance.exports.memory.buffer, ptr, 4);
// ptrValor Uint8Array(4) [ 8, 5, 8, 5 ]
console.log('ptrValor', ptrValor);

// -----
// Função sem declaração de "extern"
// e retorno do valor de forma direta.
const semPonteiro = sem_ponteiro();

console.log(semPonteiro); // 84411656

// Obviamente vamos esperar um erro na próxima linha
// pois o valor é fora dos limites da memória.
const semPonteiroValor = new Uint8Array(
  instance.exports.memory.buffer, semPonteiro, 4);
```

As primeiras duas funções usadas no construtor da matriz tipada como `Uint8` retornam os dados que queremos, porém a última função do compilador assume a responsabilidade de tratamento do retorno da função pela falta de especificação da C ABI.

Vale destacar que o compilador Rust pode retornar certos valores diretamente na ausência da C ABI e, em outros casos, como valores maiores que o máximo do `i32`, não terá retorno algum. No código a seguir, temos um exemplo de uma função que retorna um `u8` e outra função que retorna uma lista de 33 valores `u8`.

```
#[no_mangle]
fn valor() -> u8 {
   85
}

#[no_mangle]
fn valor_maior_que_i32() -> [u8; 33] {
    *b"Bem-vindo ao mundo do WebAssembly"
}
```

Se executarmos essas funções no navegador, a função que retorna o tipo `u8` retornará o valor diretamente, enquanto a outra função,

`valor_maior_que_i32`, retornará `undefined`:

```
const {
  valor,
  valor_maior_que_i32,
} = instance.exports;

// 85
console.log(valor());
// undefined
console.log(valor_maior_que_i32());
```

Sem o uso explícito de ponteiros (até mesmo com o uso da C ABI), é necessário um entendimento de como o compilador do Rust vai tratar cada caso, o que pode ficar mais complicado em diferentes cenários. Por exemplo, se quisermos receber valores pelo lado do WebAssembly usando um parâmetro de função com uma característica não dimensionável (`?Sized`), como `str` ou um `[u8]`, o valor enviado é dividido em dois parâmetros: um valor é o ponteiro para os dados, o outro valor é um ponteiro para um *bit* de metadados.

Quando criamos e usamos ponteiros como retorno no Rust, estamos basicamente salvando a memória em um bloco (essa afirmativa muda caso a caso, uma vez que você pode liberar a memória ou ter a referência do ponteiro em uma memória temporária), que será transmutada para a memória do WebAssembly.

Embora a C ABI seja frequentemente usada para o WebAssembly, não é em todos os casos que você precisa usá-la. Muitas vezes, as funções exportadas não realizam nenhum tipo de retorno ou são retornos baseados em tipos primitivos.

O nosso editor de imagens do capítulo anterior, por exemplo, é um caso que não precisaria ter sido escrito com o uso da C ABI, porém, pela compatibilidade e o fato de a C ABI não sofrer mudanças, é visto como uma boa prática pela comunidade o uso da C ABI se o(a)

programador(a) não tem o entendimento sobre como as ABIs funcionam em Rust.

Sabemos agora como uma ABI funciona em Rust e qual ABI escolher para diferentes casos. Na próxima seção, vamos ver como fazer a configuração das ABIs de forma mais declarativa.

5.2 Blocos externos

No decorrer deste capítulo, usamos a palavra-chave `extern` exclusivamente em funções, mas também é possível usá-la em blocos, o que resulta na criação de blocos externos. Portanto, blocos externos são blocos de código envelopados pelo uso da palavra-chave `extern` e, dentro deles, você pode definir múltiplas funções e valores.

Na linguagem de programação Rust, os blocos externos podem conter a definição de informações específicas úteis para o processo de vinculação fazendo uso dos atributos. Nesta seção, vamos ver o uso de dois atributos úteis em blocos externos: `link` e `link_name`.

Começaremos com o atributo `link`, que especifica o nome de uma biblioteca nativa que o compilador deve vincular para obter os itens específicos dentro de um bloco externo.

O atributo `link` possui algumas chaves que auxiliam na vinculação:

- A chave `name` é o nome da biblioteca nativa a ser vinculada;
- A chave opcional `kind` é um valor que especifica o tipo de biblioteca (`dylib`, `static` ou `framework`), onde, por padrão, é configurada como `dylib`;
- A chave opcional `modifiers` é uma forma de especificar modificadores de vinculação para vincular a biblioteca;
- A chave `wasm_import_module` é usada para especificar o nome do módulo WebAssembly dos itens dentro de um bloco externo.

> Nota importante: **o nome padrão do módulo é** `env` se `wasm_import_module` não for especificado.

Veja um exemplo de uso das chaves para diferentes casos:

```
// Usando a biblioteca C++ "snappy" desenvolvida pelo Google
(https://github.com/google/snappy)
// Atualmente, o Rust não consegue chamar uma função diretamente de uma
biblioteca C++
// porém o snappy inclui uma interface C (documentada em "snappy-c.h").
#[link(name = "snappy")]
extern {
    // mapeia a função já existente
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

// Exemplo usando o "CoreFoundation"
// É uma ABI em C desenvolvida pela Apple
#[link(name = "CoreFoundation", kind = "framework")]
extern {
    // ...
}

// Exemplo usando um módulo WASM nomeado como "meu_modulo_webassembly"
#[link(wasm_import_module = "meu_modulo_webassembly")]
extern {
    // ...
}
```

Você também pode usar o atributo `link_name`, que especifica declarações dentro de um bloco externo para indicar o nome do símbolo (função ou valor) que será importado. O código a seguir exemplifica o uso do atributo `link_name`:

```
extern {
    #[link_name = "nome_real_do_simbolo"]
    fn nome_em_rust();
}
```

Ou seja, podemos criar nomes customizáveis para os símbolos. A função `nome_real_do_simbolo` será nomeada como `nome_em_rust` dentro do contexto de execução do Rust.

Agora que vimos o básico de como os atributos de vinculação funcionam em Rust, que tal vermos na prática como usar os conceitos de vinculação e ABI em programas compilados para WebAssembly?

5.3 Compilando WebAssembly com vinculação dinâmica e estática

Vamos criar um arquivo `vinculacao.rs`, no qual vamos definir duas funções externas:

- A função `log` será vinculada à função `console_log`, que executa o `console.log` do JavaScript;
- A função `alert` executa o `window.alert` do JavaScript.

```
extern "C" {
    #[link_name = "console_log"]
    fn log(x: u32) -> u32;

    #[link_name = "alert"]
    fn alert(x: u8) -> u8;
}

#[no_mangle]
pub fn executar() {
    unsafe {
        log(20); // 20
        log(87654321); // 87654321

        alert(85); // 85 com window.alert
    }
}
```

Para compilar em WebAssembly usando esse código, vamos fazer de forma diferente. Anteriormente, no decorrer do livro, estávamos usamos o `cargo` para lidar com o processo de criação, mas também é possível usar o `rustc` (compilador do Rust). Veja o exemplo a seguir:

```
rustc vinculacao.rs --target wasm32-unknown-unknown --crate-type=cdylib
```

Esse comando é bem parecido com a forma como compilamos usando `cargo`, porém é mais prático do que criar um projeto `cargo` e suas respectivas configurações. O `rustc` recebe o arquivo `vinculacao.rs` como arquivo de entrada e retorna um arquivo de saída `vinculacao.wasm` no mesmo nível do diretório de `vinculacao.rs`.

Perceba também que especificamos o destino como `wasm32-unknown-unknown` usando `--target`. O mesmo acontece com `--crate-type`, que especificamos como `cdylib`.

No código escrito anteriormente, as funções do JavaScript serão vinculadas dinamicamente, mas também queremos ver na prática a vinculação estática. Para poder realizar de forma estática, precisamos criar outro arquivo Rust e compilar para um `staticlib`.

Criaremos uma arquivo chamado `numeros.rs` e duas funções:

- A função `retorna_10` será nomeada na lista de símbolos como `retorna_dez`;
- A função `quadrado` usa o `#[no_mangle]`, que avisa ao compilador para não realizar mudança no nome da função posteriormente.

A seguir, o código de `numeros.rs`:

```
static DEZ: i32 = 10;
#[export_name = "retorna_dez"]
pub extern fn retorna_10() -> i32 {
    DEZ
}

#[no_mangle]
```

```
pub extern fn quadrado(x: i32) -> i32 {
    x * x
}
```

A partir desse código, vamos compilar para uma `staticlib` em vez de um `cdylib`, então precisamos especificar isso no comando de compilação com `--crate-type=staticlib`. O comando para compilar `numeros.rs` é:

```
rustc numeros.rs --target wasm32-unknown-unknown --crate-type=staticlib
```

Perceba que, em vez de criar um arquivo *.wasm*, foi criado um arquivo `libnumeros.a` (se você estiver no Windows, é provável que também seja criado adicionalmente um arquivo com a extensão *.lib*). Esse arquivo contém nossa biblioteca estática com seus respectivos símbolos.

Você pode usar o comando `nm` no terminal para ver as definições dos símbolos criados em `libnumeros.a`. O comando `nm`, que vem do inglês "*name mangling*", é um comando Unix usado para listar a tabela de símbolos e seus atributos de um arquivo executável binário, incluindo bibliotecas, módulos de objetos compilados, arquivos de objetos compartilhados e executáveis autônomos.

A análise dos símbolos criados em `libnumeros.a` é um passo importante para validar se os símbolos foram injetados corretamente (por exemplo: validar os nomes das funções criadas dentro do arquivo) e, para isso, podemos executar o seguinte comando:

```
$ nm libnumeros.a
```

Esse comando vai retornar uma grande saída de texto. O resultado de saída muda com base em várias variáveis, tais como versão do compilador, sistema operacional usado, entre outros. A seguir, as primeiras oito linhas da saída de texto gerada:

```
numeros.numeros.108babcb-cgu.0.rcgu.o:
00000010 d .L__unnamed_1
00000004 d .L__unnamed_2
```

```
         U core::panicking::panic::h3b7a8a1eaf47fc0f
00000000 d numeros::DEZ::h0e16c9e24131baad
0000000d T quadrado
00000001 T retorna_dez
00000020 d str.0
```

Ufa! Os nomes estão corretos. Conseguimos ver alguns símbolos ali, um deles é `numeros::DEZ::h0e16c9e24131baad`, que reflete o valor estático `DEZ`. Outros símbolos também visíveis são as funções `quadrado` e `retorna_dez`.

Podemos presumir que a biblioteca foi compilada de forma correta por possuir os símbolos, mas não se prenda à leitura do arquivo inteiro, já que o formato do arquivo foi criado para ser processado pelo computador.

Uma vez que temos nossa biblioteca estática (`staticlib`), precisamos atualizar o código do arquivo `vinculacao.rs` :

```rust
extern "C" {
    #[link_name = "console_log"]
    fn log(x: i32) -> i32;

    #[link_name = "alert"]
    fn alert(x: i32) -> i32;
}

// Define o nome da biblioteca como "numeros"
// Também configura o tipo da biblioteca como estática
#[link(name = "numeros", kind = "static")]
extern "C" {
    fn retorna_dez() -> i32;

    fn quadrado(x: i32) -> i32;
}

#[no_mangle]
pub fn executar() {
    unsafe {
```

```
        let dez = retorna_dez();

        log(20); // 20
        log(87654321); // 87654321

        log(dez); // 10
        log(quadrado(dez)); // 100

        alert(85); // 85 com window.alert
    }
}
```

A compilação desse código pode ser realizada apenas se especificarmos ao compilador Rust onde a biblioteca `numeros` está localizada, pois é uma biblioteca estática, ou seja, precisamos dessas funções em tempo de compilação. Existem algumas opções de como fazer isso e uma delas é definir o parâmetro `-L` no compilador.

No `rustc`, o parâmetro `-L` permite adicionar um diretório no caminho de pesquisa da biblioteca realizado pelo compilador. Você também pode especificar no mesmo parâmetro dentro do diretório se você quer apenas uma dependência, *crate*, recurso nativo, *framework*, ou "todos os tipos possíveis" (que é o padrão).

O comando a seguir realiza a compilação para WebAssembly com o uso da biblioteca estática `numeros` usando o caminho `./`, que reflete na raiz do diretório onde está o arquivo da nossa biblioteca estática:

```
rustc vinculacao.rs --target wasm32-unknown-unknown --crate-type=cdylib -L ./
```

Já faz um tempinho que não escrevemos JS, mas agora temos a desculpa perfeita para isso! Precisamos configurar as funções externas (`console_log` e `alert`) no JavaScript. Lembra daquele segundo parâmetro de configuração nos métodos de instanciamento? Pois então, precisamos usar exatamente esse parâmetro para definir as importações:

```
// Define as importações
const importacoes = {
  // O nome padrão é "env"
  env: {
    console_log: (n) => console.log(n),
    alert: (n) => window.alert(n)
  }
};

fetch('vinculacao.wasm')
  .then(response => response.arrayBuffer())
  .then(bytes => WebAssembly.compile(bytes))
  // Configura as importações na instância
  .then(mod => WebAssembly.instantiate(mod, importacoes))
  .then(({exports: { executar }}) => {
    // Executa a função "executar"
    executar();
  });
```

Quando executarmos esse código no navegador, ocorrerão múltiplas execuções do `console.log` com um alerta na página de `85`.

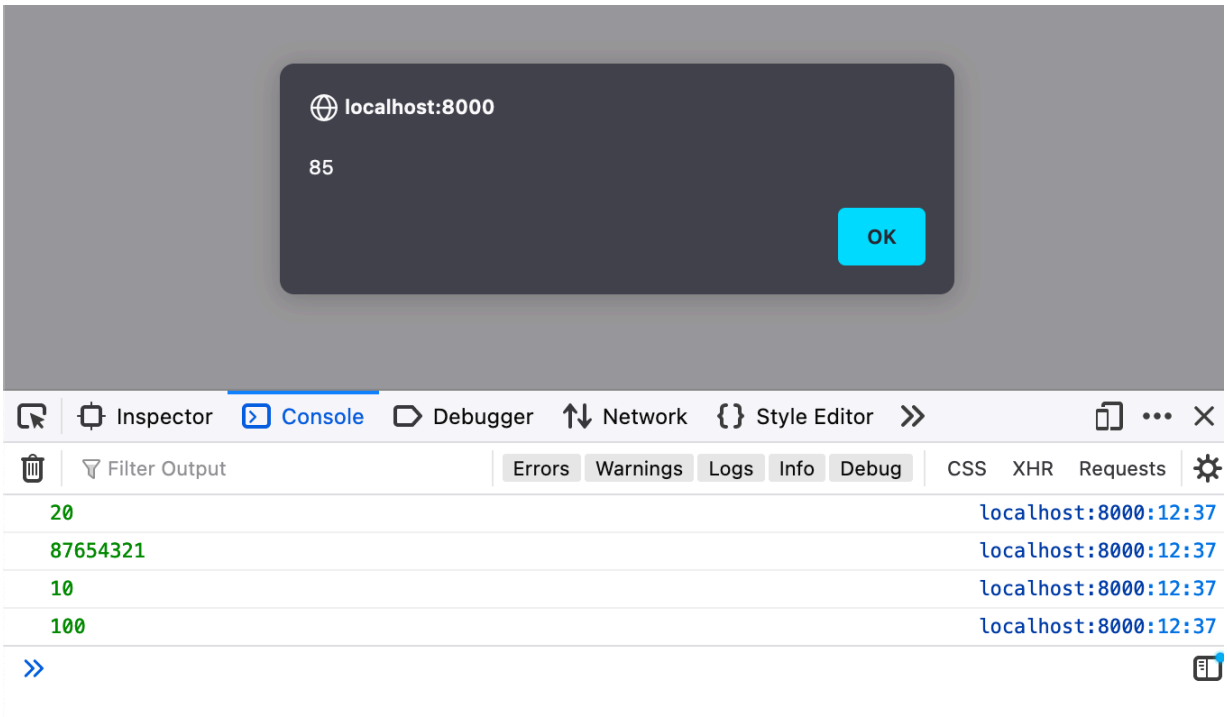


Figura 5.1: Resultado vinculação estática e dinâmica com WebAssembly.

A parte mais legal disso tudo é que é possível explorar e selecionar entre as diversas bibliotecas estáticas que existem atualmente e compilar diretamente para a Web com WebAssembly!

Além disso, vimos como escrever um código Rust que pode usar funções do JavaScript diretamente, o que permite criar inúmeras possibilidades, como manipulação do DOM, entre outras funções que o navegador provê e que podem ser diretamente executadas pelo WebAssembly como uma função externa.

No próximo capítulo, vamos aprender como fazer o uso de *strings* no WASM, aprendizado que permite expandir inúmeras possibilidades em relação ao desenvolvimento de aplicações com WebAssembly.

Capítulo 6
Encoding e Strings

O WebAssembly, como vimos anteriormente, suporta os seguintes tipos primitivos:

- Números inteiros de 32 bits;
- Números inteiros de 64 bits;
- Números em ponto flutuante (*float*) de 32 bits;
- Números em ponto flutuante (*float*) de 64 bits.

A escassez de opções de tipos primitivos torna inviável a comunicação de forma nativa do WASM usando diferentes estruturas de dados como vetores, *strings*, estruturas baseadas em chave e valor, entre outras.

Entretanto, a falta de opções nos tipos primitivos não é um problema pela possibilidade de compartilhar memória entre JavaScript e WebAssembly. O compartilhamento de memória viabiliza o uso de matrizes tipadas e torna fácil a comunicação de qualquer tipo de dados bruto, como a de uma imagem, que vimos no capítulo 4.

A leitura e escrita de dados brutos na memória compartilhada abre inúmeras portas. Uma delas é o uso de tipos além dos que o WASM provê, por exemplo, uma *string*. Neste capítulo, nosso objetivo é aprender sobre o uso de *strings* no WASM.

Como foi possível usar uma *string* no início do livro?

As *strings* não fazem parte das estruturas de dados que são suportadas nativamente pelo WebAssembly. Isso significa que podem ser classificada como uma estrutura de dados não primitiva, criada pelo(a) programador(a) e que não é predefinida pelo WebAssembly.

Lembra do módulo que escrevemos no capítulo 2 usando `wasm-bindgen` para criar e retornar a mensagem "olá mundo"? Quando escrevemos o módulo "olamundo" com o `wasm-bindgen`, vimos que a ferramenta é responsável pela criação do módulo e instanciamento do WebAssembly de forma automática. O código gerado pelo `wasm-bindgen` é baseado no código escrito em Rust. A produção de forma automática do código JavaScript que lida e gerencia os módulos é uma grande vantagem em termos de produtividade, mas dificulta entender o que, de fato, acontece por trás dos panos.

É importante entender como lidar com diferentes tipos de estrutura de dados e, para isso, precisamos aprender um conceito amplamente utilizado na computação conhecido como codificação e decodificação (*Encoding and Decoding*, em inglês) de dados.

6.1 Codificação e decodificação de dados

Na Ciência da Computação, a codificação de dados é o processo de colocar uma sequência de caracteres (letras, números, pontuação e símbolos) em um formato especializado para transmissão ou armazenamento eficiente.

> Esses termos não devem ser confundidos com criptografia e descriptografia, que se concentram em ocultar e proteger dados. Podemos criptografar dados sem alterar o código ou codificar dados sem ocultar deliberadamente o conteúdo.

Os termos codificação e decodificação são frequentemente usados em referência aos processos das conversões analógico-digital e digital-analógico. Nesse sentido, esses termos podem se aplicar a qualquer forma de dados, incluindo texto, imagens, áudio, vídeo, multimídia, programas de computador ou sinais em sensores, telemetria e sistemas de controle.

Um exemplo de codificação é a de vídeo (*Video Encoding*). A codificação de vídeo é o processo de conversão de dados não processados do sensor de imagem de uma câmera (um arquivo RAW, do inglês, *Raw Image Format*) em arquivos digitais para que eles não sejam salvos como imagens individuais, mas como vídeos fluidos. É certo assumir que qualquer programa de edição ou reprodução de vídeos depende da codificação e decodificação de dados.

A internet também depende da codificação. Uma URL (acrônimo em inglês para *Uniform Resource Locator*), o endereço de uma página da Web, só pode ser enviada pela internet usando o *American Standard Code for Information Interchange* (também conhecido como ASCII), que é o código usado para arquivos de texto em computação.

O sistema de codificação ASCII usa exatamente 8 bits por caractere. No ASCII, um número binário representa um caractere, que pode ser letras maiúsculas ou minúsculas, números, sinais de pontuação e outros símbolos comuns.

O caractere é escrito de 0 e 1, por exemplo: "01000101" ("E" escrito com maiúscula) ou "01101100" ("l" escrito com minúsculo). Com isso, permitem 256 valores que usam 0 e 1 em cada uma das oito casas, indo de "00000000", depois "00000001" até "11111111".

CURIOSIDADE: as URLs não podem conter espaços e geralmente têm caracteres que não estão no conjunto de caracteres ASCII. A codificação de URL, também conhecida como codificação de porcentagem (em inglês, *Percent-encoding*), resolve isso por meio da conversão de espaços em um sinal `+` ou com o texto "`%20`", justamente da conversão de caracteres não ASCII em um formato ASCII válido.

Mas não se engane, a codificação ASCII é mais antiga que a internet. A ASCII foi desenvolvida a partir do código do telégrafo.

Seu primeiro uso comercial foi como um código de teleimpressor de 7 bits. As regras de codificação passaram por diversas revisões ao longo dos anos.

A seguir, uma imagem histórica da tabela ASCII de uma impressora manual pré-1972:

USASCII code chart

Bits b_7 b_6 b_5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4	b_3	b_2	b_1	Row ↓ Column →	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Figura 6.1: Tabela ASCII. Fonte: Domínio público.

O uso do formato ASCII para operações de intercâmbio em rede foi descrito em um documento oficial em 1969, e esse documento foi formalmente utilizado nos padrões da internet em 2011.

Existem também outros formatos de codificação e decodificação comumente usados na programação além do ASCII, como BinHex, MIME (acrônimo do inglês: *Multipurpose Internet Mail Extensions*), Unicode e outros.

Mas por que a história do ASCII é importante?

Quando o ASCII foi criado, os caracteres foram representados através do alfabeto do inglês norte-americano e na época era em 7 bits. Isso significa que, em vez de 256 valores possíveis, eram apenas 128. Os valores eram: números de zero a nove, caracteres do alfabeto norte-americano e alguns caracteres de controle. Posteriormente, o ASCII evoluiu para o uso de 8 bits e diversas empresas começaram a preencher os 128 espaços vazios de sua própria forma. Em razão disso, surgiram diversos problemas na internacionalização dos dados por parte da codificação e decodificação.

Por exemplo, um computador comprado no Brasil deve exibir as letras acentuadas da língua portuguesa, como "ê" ou "é". No entanto, esses caracteres em um computador adquirido em outro país exibem as letras de forma diferente e, consequentemente, a palavra formada com esse caractere era representada erroneamente.

Para resolver esse tipo de problema, surgiu o ANSI (do inglês *American National Standards Institute*), que mantém a tabela ASCII, e define que cada país tenha uma página de código com a utilização dos 128 excedentes.

Outra curiosidade: o número oficial da página da língua portuguesa é 860, que também suporta italiano e espanhol. Entretanto, no Brasil, o número da página que ganhou adesão popular em todo o país foi o 850.

No entanto, ainda não era possível representar caracteres asiáticos em 8 bits. Para resolver esse problema, foi criado o conjunto de caracteres de byte duplo (DBCS). O DBCS é uma codificação de caracteres na qual todos (incluindo caracteres de controle) são codificados em dois bytes, ou apenas todos os caracteres gráficos não representáveis por um conjunto de caracteres de byte único (SBCS) são codificados em dois bytes (caracteres Han geralmente compreendem a maioria desses caracteres de dois bytes).

Um DBCS suporta idiomas nacionais que contêm muitos caracteres ou símbolos exclusivos, dado que o número máximo de caracteres que podem ser representados com um byte é 256 caracteres, enquanto dois bytes podem representar até 65.536 caracteres.

Outra solução para os problemas decorrentes da internacionalização de caracteres foi a criação do Unicode, que também é dividido em páginas, sendo que, para cada página, existe uma sequência de caracteres, cada um representado por uma sequência de bits — ou seja, basicamente, há um identificador para cada caractere.

Para a palavra "Wasm", por exemplo, temos a seguinte representação: `U+0057 U+0061 U+0073 U+006D` . No entanto, os norte-americanos raramente usavam pontos de código maiores que `U+00FF` e, por isso, inventaram o UTF (forma de transformação Unicode). Baseado no UTF, surgiram diferentes formas: o UTF-8, UTF-16 e UTF-32, que são diferentes, porém codificam para o mesmo padrão Unicode.

Das diferentes formas de transformação criadas para atender o Unicode, o UTF-8 ganhou maior relevância e popularidade ao longo dos anos.

6.2 UTF-8

Em 2022, o UTF-8 é a codificação dominante para a *World Wide Web* e tecnologias da internet, representando 98% de todas as páginas da Web e até 100% para diversos idiomas. Muitos dos padrões relacionados com a internet suportam apenas UTF-8, por exemplo, no uso de JSON para troca de dados sem uma marca de ordem de byte (*BOM*). Também é a recomendação do WHATWG para especificações HTML e DOM. O Internet Mail Consortium também recomenda que todos os programas de e-mail sejam capazes de exibir e criar mensagens usando UTF-8.

O UTF-8 é capaz de codificar todos os 1.112.064 pontos de código de caracteres válidos em Unicode usando de uma a quatro unidades de código de 1 byte (8 bits). Os pontos de código com valores numéricos mais baixos, que tendem a ocorrer com mais frequência, são codificados usando menos bytes.

Ele foi projetado para compatibilidade com versões anteriores com ASCII: os primeiros 128 caracteres de Unicode, que correspondem um a um com ASCII, são codificados usando um único byte com o mesmo valor binário do ASCII, para que o texto ASCII seja válido no UTF-8 também. A compatibilidade e o *fallback* do UTF-8 tornaram o modelo de codificação extremamente popular.

> Descrições e fórmulas do UTF-8 podem ser encontradas no Anexo D da ISO/IEC 10646-1 (ISO.10646).

Os pontos de código são codificados no UTF-8 de um a quatro bytes, dependendo do valor do ponto de código. Na tabela a seguir, os caracteres x são substituídos pelos bits do ponto de código:

Ponto de código ↔ conversão para UTF-8

Primeiro ponto de código	Último ponto de código	Byte 1	Byte 2	Byte 3	Byt
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xx

Na tabela, você pode ver que os primeiros 128 pontos de código (abrange o ASCII) precisam apenas de um byte. Os próximos 1.920 pontos de código precisam de dois bytes para codificar, o que abrange o restante de quase todos os alfabetos de escrita latina (com as extensões IPA), alfabetos grego, cirílico, copta, armênio, hebraico, árabe, siríaco, N'Ko e Tāna.

Podemos concluir que o UTF-8 é altamente confiável nas questões de internacionalização de dados por suportar todas as linguagens, além do fato de ser utilizado por quase todos os sites da internet e definido como a opção padrão na maioria das ferramentas de desenvolvimento Web. Pela totalidade dos fatores descritos, o UTF-8 se tornou o formato das *strings* no Rust.

6.3 Strings do Rust para WASM

Vamos primeiro definir o que queremos dizer com o termo *string*. Rust tem apenas um tipo de *string* na linguagem principal, que é a fatia de *string* `str`, geralmente vista em sua forma emprestada `&str`. Podemos visualizar um `&str` como `&[u8]`.

O tipo `String`, que é fornecido pela biblioteca padrão do Rust em vez de codificado na linguagem principal, é um tipo de *string* codificado em UTF-8 que pode ser aumentado, mutável e de propriedade. Dado que é possível modificar o tamanho da *string*, podemos visualizá-lo como um `Vec<u8>`.

Uma *literal string* é um `&str` que faz referência a um texto pré-alocado, normalmente armazenado na memória, que permite apenas a leitura, junto ao código de máquina do programa. Os *bytes* são criados quando o programa inicia a execução e duram até o final do programa. Portanto, é impossível modificar um `&str`.

O `&str` pode se referir a qualquer fatia de qualquer *string*, portanto, é apropriado usar `&str` como parte dos argumentos da função e, desta forma, a função pode receber também o tipo `String`, como no exemplo a seguir:

```
fn retorna_str(texto: &str) -> &str {
    texto
}

let tipo_string = String::from("string");
retorna_str(&tipo_string); // "string"

let tipo_str = "str";
retorna_str(tipo_str); // "str"
```

Novas *strings* podem ser criadas em tempo de execução, usando `String`. Uma *literal string* pode ser convertida em uma `String` usando os métodos `to_string()` e `String::from`.

```
let s = "olá".to_string();
let s = String::from("olá");
```

Para converter uma `String` em números, basta usar o método `as_bytes`. O código Rust pega a *string* `"wasm"` e a converte em *bytes* e, em seguida, imprime as duas versões da *string* no terminal.

```rust
let wasm: String = String::from("wasm");

println!("String: {:?}", &wasm);
// String: "wasm"

println!("Bytes: {:?}", &wasm.as_bytes());
// Bytes: [119, 97, 115, 109]
```

A *string* `"wasm"` retorna os bytes `119, 97, 115, 109`. Sabemos que o tipo `String` no Rust é codificado e decodificado em UTF-8, então podemos assumir que, para transformar em `String` de novo, seria possível apenas utilizar os bytes.

O tipo `String` provê um método chamado `from_utf8` que permite criar *strings* em tempo de execução usando um vetor de `u8`. O método `from_utf8` não consegue prever o tamanho fixo do *array* e por isso especifica que deseja receber um vetor em vez de um *array*. Se você quiser os dados de um *array*, é necessário a conversão para vetor usando o método `.to_vec`.

```rust
let wasm_bytes: Vec<u8> = vec![119, 97, 115, 109];

let wasm_texto: String = String::from_utf8(wasm_bytes).unwrap();

println!("resultado da conversão: {:?}", wasm_texto);
// resultado da conversão: "wasm"
```

Incrível! Agora sabemos como converter *bytes* para *string* e fazer a operação reversa também no Rust. Como seria possível fazer a codificação e decodificação no WebAssembly?

6.4 Envio de Strings do Rust para o WebAssembly

Lembrete: se você tiver interesse em reproduzir o exemplo de *string* deste capítulo, basta acessar este link: https://github.com/raphamorim/livro-webassembly-exemplos.

Nesta etapa, enviaremos os dados de uma *string* UTF-8 criada pelo Rust e compilada em WASM como bytes. Para isso, precisamos escrever nosso arquivo Rust que contêm a *string*. Nomeie o arquivo como *livro.rs*. No arquivo *livro.rs*, vamos escrever duas funções:

- `meu_livro` : função que retorna apenas a *string* do conteúdo do livro que desejamos retornar no navegador;
- `salvar_livro_na_memoria` : função que salva os *bytes* do conteúdo do livro nas primeiras posições da memória padrão do WebAssembly.

Primeiro, vamos começar escrevendo a função `meu_livro` . Na função a seguir, reutilizei os primeiros parágrafos da obra literária brasileira *Vidas Secas*, de Graciliano Ramos, publicada em 1938, porém sinta-se à vontade para preencher como desejar.

livro.rs

```
// Para compilar o arquivo:
// rustc livro.rs --target wasm32-unknown-unknown --crate-type=cdylib

extern crate core;

use std::ptr;
use core::slice::from_raw_parts_mut;

fn meu_livro() -> String {
    // trecho retirado da segunda edição do livro "Vidas Secas" de
Graciliano Ramos
    String::from("Na planície avermelhada os juazeiros alargavam duas
manchas verdes. Os infelizes tinham caminhado o dia inteiro, estavam
cansados e famintos. Ordinariamente andavam pouco, mas como haviam
repousado bastante na areia do rio seco, a viagem progredira bem três
```

```
léguas. Fazia horas que procuravam uma sombra. A folhagem dos juazeiros
apareceu longe, através dos galhos pelados da catinga rala. Arrastaram-se
para lá, devagar, sinhá Vitória com o filho mais novo escanchado no quarto
e o baú de folha na cabeça, Fabiano sombrio, cambaio, o aió a tiracolo, a
cuia pendurada numa correia presa ao cinturão, a espingarda de pederneira
no ombro. O menino mais velho e a cachorra Baleia iam atrás.")
}
```

Nesse código, criamos a *string* do livro; precisamos agora salvar os *bytes* do livro na memória, para podermos fazer a leitura no lado do JavaScript depois. A seguir, veja o código da função `salvar_livro_na_memoria`:

livro.rs

```
#[no_mangle]
pub fn salvar_livro_na_memoria() {
    let fatia_da_memoria: &mut [u8];
    let ponteiro_nulo: *mut u8 = ptr::null_mut();

    // Conteúdo do livro
    let livro: String = meu_livro();
    // Bytes do conteúdo do livro
    let bytes: &[u8] = livro.as_bytes();

    unsafe {
        // Define a fatia memória do ponteiro nulo até o tamanho dos dados
do livro
        fatia_da_memoria = from_raw_parts_mut::<u8>(ponteiro_nulo,
bytes.len());
    }

    // Percorre os _bytes_ do livro e salva na memória
    for pos in 0..bytes.len() {
        fatia_da_memoria[pos] = bytes[pos];
    }
}
```

Perfeito! Após escrever essa função, fechamos a parte do Rust e WebAssembly. Agora precisamos escrever o código no lado do

navegador para carregar o arquivo *livro.wasm*.

O primeiro passo é escrever o arquivo *index.html*, que carrega o *livro.js* e também adiciona estilos simples na página:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Livro</title>
  <style>
    body {
      background: beige;
      font-family: monospace;
      padding: 20px;
    }
  </style>
</head>
<body>
  <div id="conteudo"></div>
  <script src="livro.js"></script>
</body>
</html>
```

Após isso, criamos o arquivo *livro.js*, que é responsável pelo carregamento e execução do WebAssembly. Nesse arquivo, vamos ler a memória do WASM e executar um `console.log` para ver se, de fato, temos os dados do texto no lado do navegador.

```
WebAssembly.instantiateStreaming(fetch('livro.wasm'))
.then(({ instance }) => {
  const {
    salvar_livro_na_memoria,
    memory
  } = instance.exports;

  // Insere os bytes nas posições do tamanho do texto
  salvar_livro_na_memoria();
```

```
  // Aqui lemos os primeiros 700 blocos da memória
  const memoria = new Uint8Array(memory.buffer, 0, 700);
  console.log(memoria);
});
```

Se olharmos agora a memória que foi exibida no lado do navegador, podemos conferir que os dados foram salvos corretamente. Outro ponto interessante é que também definimos o começo da visualização da memória do ponto inicial até o limite de 700 blocos. Se o limite não for definido, serão adicionados dados que não queremos ter na memória.

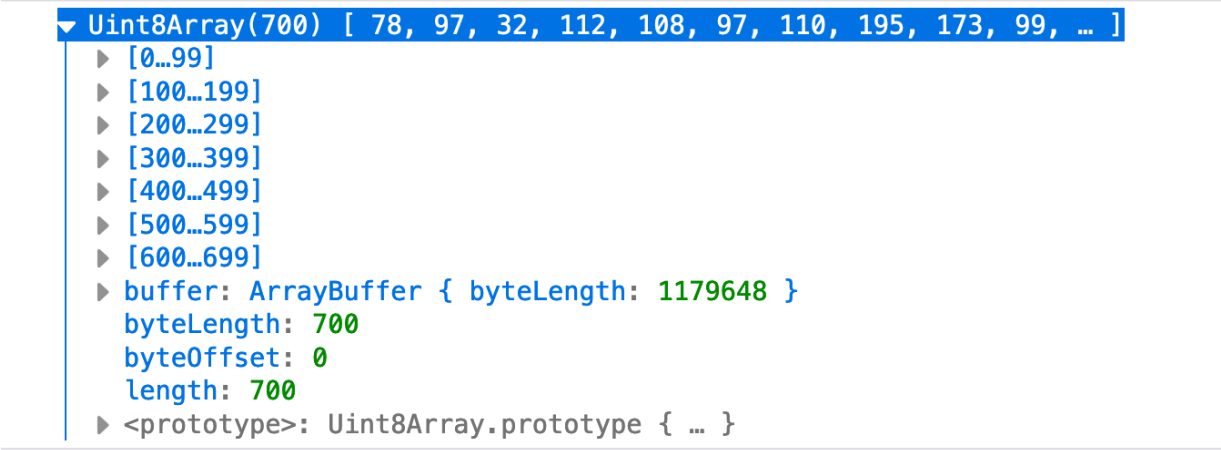


Figura 6.2: Bytes no navegador.

Estamos quase lá! Agora que temos os bytes no navegador, precisamos realizar a codificação, porém como funciona a codificação de dados para *strings* no JavaScript?

O JavaScript provê duas APIs para lidar com codificação e decodificação de texto: `TextEncoder` e `TextDecoder` (ambas possuem como padrão a codificação UTF-8). A seguir, veja o exemplo do uso das respectivas APIs:

- O `TextEncoder` é responsável pela codificação. Neste exemplo, usamos o método `encode`, que recebe um fluxo de pontos de código como entrada e emite *bytes* em UTF-8:

```
const codificador = new TextEncoder();

codificador.encode("wasm");
// Uint8Array(4) [ 119, 97, 115, 109 ]
```

- O `TextDecoder` é responsável pela decodificação dos bytes. Neste exemplo, usamos o método `decode`, que retorna uma *string* contendo texto decodificado do *buffer*:

```
const wasm = Uint8Array.from([ 119, 97, 115, 109 ]);
const decodificador = new TextDecoder();

decodificador.decode(wasm);
// "wasm"
```

Já que possuímos os *bytes*, precisamos apenas fazer a decodificação para *string*, então vamos usar o `TextDecoder` no código do arquivo *livro.js*, como no exemplo a seguir:

```
// ...

// Aqui lemos os primeiros 700 blocos da memória
const memoria = new Uint8Array(memory.buffer, 0, 700);
console.log(memoria);

// Decodifica os _bytes_ da memória para string
const decoder = new TextDecoder();
const texto = decoder.decode(memoria);

// Seleciona o elemento "#conteudo"
const conteudo = document.querySelector('#conteudo');

// Define o conteúdo de texto do elemento para o conteúdo do livro
conteudo.textContent = texto;
```

Pronto! Com esse código, acabamos de realizar a última etapa. Que tal visualizarmos se tudo funcionou corretamente no navegador?

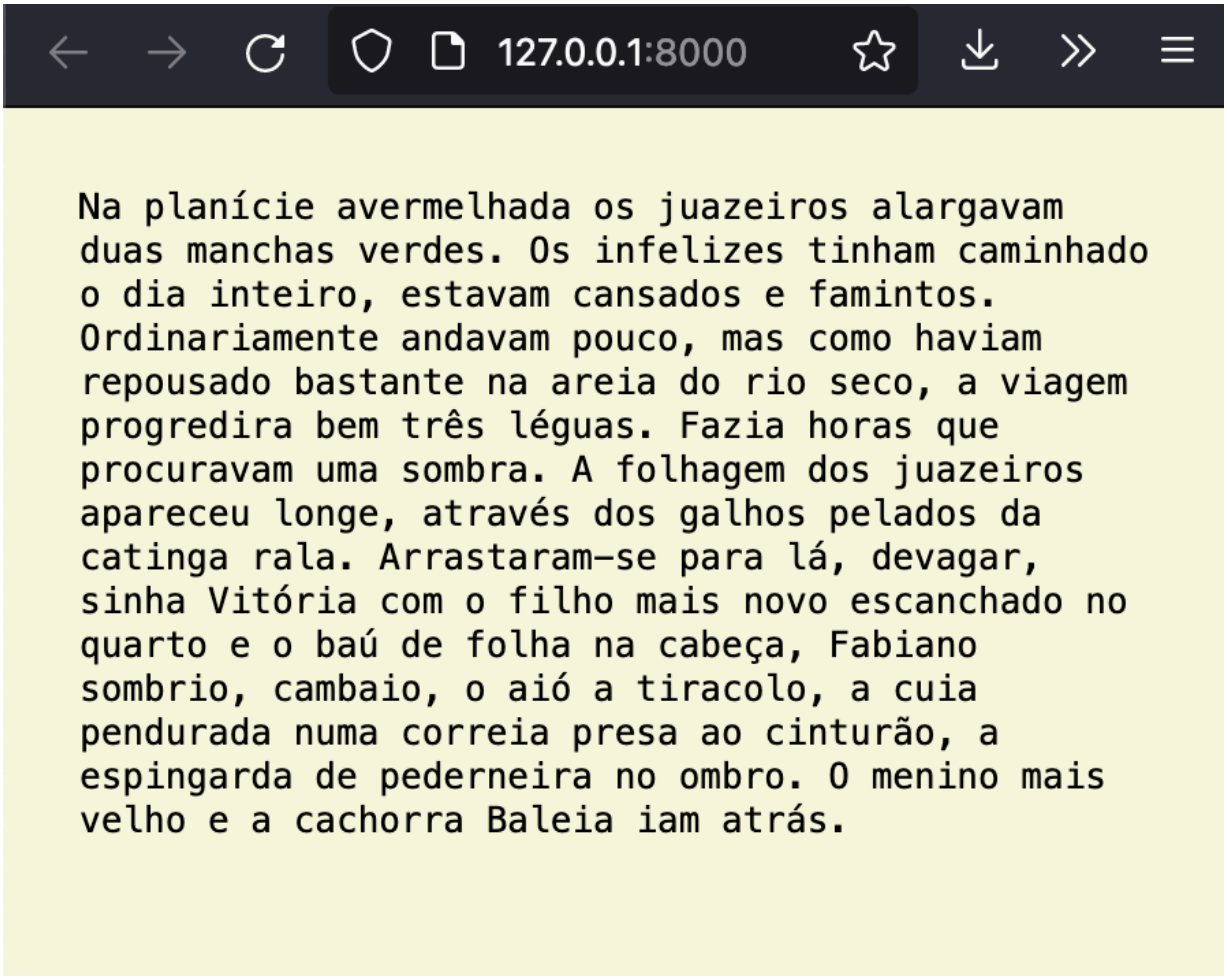


Figura 6.3: Livro.

Conforme podemos ver na imagem, a codificação e decodificação funcionaram corretamente. A *string* foi codificada no lado do WebAssembly e decodificada no JavaScript. A operação inversa também é possível usando a mesma lógica que vimos no capítulo 4 com alocação de memória no lado do JavaScript, quando salvamos os dados da imagem.

Existem algumas vantagens bem conhecidas quanto ao uso de *strings* no WebAssembly:

- O WebAssembly não permite o inspecionamento de código como o JavaScript, sendo correto assumir que validações sensíveis de *strings* no lado do WASM são mais seguras;

- O processamento de grandes textos é feito de forma rápida e performática no WASM.

Podemos concluir que o aprendizado do uso de *strings* cria diversas possibilidades, como análise, validação, edição e qualquer tipo de processamento de dados de *strings* no lado do WebAssembly.

No próximo capítulo, vamos aprender sobre como a programação paralela e concorrente funciona no WebAssembly, vendo tópicos como *threads*, atômicos, memória compartilhada e outros.

CAPÍTULO 7
Threads e atômicos

Antes de começarmos, vamos combinar que toda vez que o termo "agente" aparecer neste capítulo, ele está se referindo ao termo criado pela ECMAScript. Um agente compreende um conjunto de contextos de execução ECMAScript, uma pilha de contextos de execução, um contexto de execução em execução, um registro de agente e uma *thread* em execução. Exceto para o encadeamento em execução, os constituintes de um agente pertencem exclusivamente a esse agente.

Quando o WebAssembly foi lançado, era apenas um produto mínimo viável. Ele foi introduzido nos navegadores com o menor investimento possível a fim de testar funcionalidades e possibilidades antes de um amplo investimento. Entretanto, o WASM sempre teve planos ambiciosos para o futuro. É correto afirmar que, desde o início, sempre houve a intenção de estender a especificação do WebAssembly com novos recursos e funcionalidades.

Desde então, diversos recursos e funcionalidades foram introduzidos e, por consequência, as atualizações na especificação permitiram evoluir o WebAssembly em termos de maturidade para atender a diferentes demandas por parte de aplicações nativas que foram portadas para a Web. Ao decorrer dos anos, inúmeras propostas foram adotadas pela comunidade e tornaram-se padrões, como importação e exportação de valores globais mutáveis (quando você especifica a chave `mutable` como verdadeira dentro da configuração do `WebAssembly.Global`), mudanças no volume da memória, entre tantas outras que foram incorporadas na plataforma.

Entre as diversas propostas de atualização da especificação do WebAssembly dos últimos anos, uma das propostas que sempre teve notabilidade e popularidade entre os desenvolvedores era **o suporte**

a *threads*. Essa proposta foi implementada e introduzida no final de 2019 na septuagésima versão do navegador Google Chrome.

> Nota: se você tiver interesse em ler a documentação original da proposta, ela está disponível no GitHub em inglês: https://github.com/WebAssembly/threads

O suporte a *threads* do WebAssembly foi uma das adições de desempenho mais importantes ao WebAssembly. Isso permite que você execute partes de seu código em paralelo com núcleos distribuídos ou o mesmo código para diversos dados de entrada, dimensionando-o para quantos núcleos o usuário tiver (o que reduz significativamente o tempo de execução geral).

Muitos desenvolvedores tinham uma imagem equivocada da proposta de *threads* no WASM, como a de que seria simplesmente adicionar "pthreads" (*POSIX Threads*) ou de que funcionaria incorporada à linguagem, como a biblioteca padrão de *threads* do Rust, `std::thread`, estaria sincronizada com as *threads* do WebAssembly.

Porém, a implementação de *threads* no WebAssembly é bem diferente! Em vez de fornecer uma experiência de biblioteca completa, a proposta de *threads* especifica os blocos de construção fundamentais sobre os quais você pode construir uma biblioteca de *threads*.

> Lembrete: se você tiver interesse em reproduzir os exemplos de *thread* deste capítulo, todos os exemplos do livro (incluindo os arquivos binários) estão neste link: https://github.com/raphamorim/livro-webassembly-exemplos.

A proposta introduziu o suporte à memória linear compartilhada e algumas novas operações para acesso à memória atômica. A responsabilidade de criar e unir *threads* é delegada ao programador

ou à programadora responsável pela integração. Neste capítulo, vamos ver como a integração funciona e, para isso, precisamos começar vendo as peças que montam o sistemas de *threads* no WebAssembly:

- Memória compartilhada;
- Instruções atômicas;
- *Web Workers*.

7.1 Memória compartilhada

A memória no WebAssembly sempre funcionou baseada no modelo linear, ou seja, a memória aparece para o programa como um único espaço de endereçamento contíguo. A API do WebAssembly estendeu a configuração da memória e adicionou o suporte ao uso do `SharedArrayBuffer`. Isso permite que a memória seja compartilhada quando a configuração `shared` é verdadeira, como neste exemplo:

```
// Retorna um SharedArrayBuffer em vez do ArrayBuffer
const memory = new WebAssembly.Memory({
  initial: 10, // Memória inicial de 640 KiB
  maximum: 100, // Memória máxima de 6.4 MiB
  shared: true, // Configurada como compartilhada
});
```

O uso do `SharedArrayBuffer` permite compartilhar memória entre os agentes e a *thread* principal, ou seja, torna-se mais barato calcular uma grande quantidade de dados e enviá-los para outra *thread* e, por sua vez, elimina-se a necessidade de copiar dados entre *threads*.

Até então, apenas usamos a memória no WebAssembly e fazemos a leitura da memória no JavaScript. Porém, para o uso de *threads*,

precisamos especificar como a memória funciona. Para criar a memória via JavaScript, podemos usar o construtor `WebAssembly.Memory()` dentro do objeto de importações. A configuração da memória é lida por padrão pelo `wasm32-unknown-unknown` com a chave nomeada como `memory` dentro de `env`.

```
WebAssembly.instantiateStreaming(fetch(arquivo), { env: { memory } })
    ...
```

A segunda peça do sistema de *threads* no WASM são as instruções atômicas.

7.2 Instruções atômicas

As instruções atômicas foram introduzidas na especificação do WASM após a implementação da proposta de *threads*. Essas instruções representam os operadores atômicos de "leitura-modificação-gravação" (RMW, do inglês, *read-modify-write*), que leem atomicamente o valor de um endereço, modificam o valor e armazenam o valor resultante no mesmo endereço. Todos os operadores RMW retornam o valor lido da memória antes da execução da operação de modificação.

A seguir, alguns exemplos de operadores atômicos RMW no WASM:

Nome	Escrita	Modificação	Retorno
`i32.atomic.rmw8.add_u`	1 byte	Adição independente de sinal de 8-bits	`i8` para `i32`
`i32.atomic.rmw16.add_u`	2 bytes	Adição independente de sinal de 16-bits	`i16` para `i32`

Nome	Escrita	Modificação	Retorno
`i32.atomic.rmw.add`	4 bytes	Adição independente de sinal de 32-bits	`i32`
`i32.atomic.rmw8.sub_u`	1 byte	Subtração independente de sinal de 8-bits	`i8` para `i32`
`i32.atomic.rmw16.sub_u`	2 bytes	Subtração independente de sinal de 16-bits	`i16` para `i32`
`i32.atomic.rmw.sub`	4 bytes	Subtração independente de sinal de 32-bits	`i32`

Curiosidade: o JavaScript também provê uma API de operadores atômicos para células da memória compartilhada, incluindo também funções que permitem que os agentes esperem e despachem eventos primitivos. Por exemplo, assim como as operações de adição atômica no WASM, o JavaScript também possui o `Atomics.add`, que recebe uma matriz tipada, um índice e um valor.

Com a implementação das operações atômicas no WebAssembly, a barreira da atomicidade para compiladores de diferentes linguagens foi quebrada, resultando no suporte nativo de tipos atômicos para WASM.

Os tipos atômicos viabilizam a comunicação de memória compartilhada primitiva entre *threads* e também atuam como blocos de construção para outros tipos. Na linguagem Rust, os tipos atômicos estão localizados na biblioteca padrão da linguagem: o `std::sync::atomic`. A biblioteca de atômicos do Rust contém o

`AtomicU8`, `AtomicU16`, `AtomicU32` e outros tipos que compilam para operações atômicas.

> **Nota:** os atômicos do Rust atualmente seguem as mesmas regras para atômicos da versão 20 do C++ (especificamente o `atomic_ref`).

As variáveis atômicas são seguras para compartilhar entre *threads*, já que elas implementam o traço `Sync`, porém não fornecem o mecanismo de compartilhamento e seguem o modelo de *threading* do Rust. Atualmente, a maneira mais comum de compartilhar uma variável atômica é colocá-la em um `Arc` (um ponteiro compartilhado com contagem de referência atômica). Por exemplo:

```
let valor = Arc::new(AtomicUsize::new(85));
```

Nesse exemplo, criamos um tipo inteiro que pode ser compartilhado entre *threads* de forma segura dentro de um `Arc`. O `AtomicUsize` tem a mesma representação na memória que o tipo inteiro subjacente: `usize`. Rust torna bem simples realizar modificações atômicas em um `Arc` que contém um tipo atômico, como `Arc<AtomicUsize>`:

```
use std::sync::Arc;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::thread;

fn main() {
    let mut atomico = AtomicUsize::new(5);
    assert_eq!(*atomico.get_mut(), 5);
    *atomico.get_mut() = 8;
    assert_eq!(atomico.load(Ordering::SeqCst), 8);

    // Compartilha o atômico mutável
    let valor: Arc<AtomicUsize> = Arc::new(atomico);

    // Vai mostrar valores começando de 8 em diferentes threads
    // Os valores mudam a cada execução
```

```
    for _ in 0..10 {
        let val = Arc::clone(&valor);

        thread::spawn(move || {
            // Incrementa o valor e retorna o valor anterior.
            let v = val.fetch_add(1, Ordering::SeqCst);
            println!("{v:?}");
        });
    }
}
```

Esse exemplo demonstra como funciona o uso de *threads* com tipos atômicos em Rust. Porém, estamos usando bibliotecas padrões do Rust de atômicos para criar lógica atômica, e esse código não será refletido da forma que queremos no WebAssembly. Para trabalhar com atomicidade no WebAssembly atualmente com Rust, é preciso entender as operações atômicas do WASM.

As operações atômicas do WebAssembly permitem realizar diferentes níveis de sincronização, mas a sincronização completa geralmente requer o bloqueio real de um encadeamento até que outro seja concluído. Duas operações que auxiliam a sincronização completa são `i32.atomic.wait` e `atomic.notify`.

Com o uso da instrução `i32.atomic.wait`, podemos bloquear atomicamente uma *thread* e, em seguida, outra *thread* pode executar `atomic.notify` para ativar uma *thread* bloqueada no mesmo endereço.

Ok, mas como fazemos uso dessas instruções no WebAssembly, uma vez que não temos nem o poder para criar *threads* com a biblioteca padrão da linguagem? Para responder a essa pergunta, vamos para a última etapa do quebra-cabeça de *threads* do WASM.

7.3 Web Workers

A última parte de como as *threads* funcionam no WebAssembly são os *Web Workers*. Um dos pontos fortes do WebAssembly é que ele estende a plataforma Web em vez de tentar substituí-la. Embora os módulos WASM manipulem números diretamente, eles podem importar qualquer função arbitrária que permite acesso total ao WASM à plataforma Web, assim como vimos anteriormente usando vinculação dinâmica.

O comitê do WebAssembly coloca foco na reutilização e aprimoramento da experiência da plataforma Web, evitando a necessidade de reinventar a roda para novas funcionalidades e, por esse motivo, o WebAssembly, quando criou o suporte a *threads*, basicamente deu aos desenvolvedores as peças em vez de uma solução pronta. Em outras palavras, a pessoa programadora é responsável por definir como as *threads* vão funcionar.

Nesta seção vamos ver isso na prática: montar um sistema de *threads* do WebAssembly baseado em *Web Workers*.

Os *Web Workers* são um meio simples para o conteúdo da Web executar scripts em *threads* em segundo plano. A *thread*, nomeada como *worker* (no português, "trabalhador"), pode executar tarefas sem interferir na interface do usuário. Além disso, eles podem executar operações de I/O usando `XMLHttpRequest` (embora os atributos `responseXML` e `channel` estejam sempre nulos) ou usando `fetch`. Uma vez criado, um *worker* pode enviar mensagens para o código JavaScript que o criou e vice-versa.

Criar um *worker* é simples: tudo o que você precisa fazer é chamar o construtor do `Worker` especificando a URL de um *script* para executar:

```
// Cria o worker baseado no script: meu-script.js
const meuWorker = new Worker('meu-script.js');

// Você também pode terminar a execução de um worker usando o método
"terminate"
meuWorker.terminate();
```

Esse código cria um *worker* e termina sua execução, mas podemos dar vida ao *worker* escutando e enviando eventos, como nos códigos a seguir:

app.js

```
meuWorker.addEventListener('message', e => {
    console.log('chegou a mensagem: ', e);
}, false);

meuWorker.postMessage('primeira mensagem');
meuWorker.postMessage('segunda mensagem');
meuWorker.postMessage('terceira mensagem');
meuWorker.postMessage('quarta mensagem');
```

worker.js

```
addEventListener('message', async (e) => {
  postMessage(e.data);
}, false);
```

Baseado nesses códigos, o resultado deve ser retornar as mensagens, porém perceba que nem sempre elas retornam em ordem. Isso se deve ao fato de serem executadas concorrentemente em diferentes *threads*. Nosso objetivo é tirar proveito das *threads* para serem executadas de forma paralela no navegador.

Computação paralela no navegador

A computação paralela foca em realizar operações ao mesmo tempo, sob o princípio de que uma tarefa pode ser dividida em subtarefas que são resolvidas paralelamente. Historicamente, os programas de computadores foram escritos para serem processados de forma sequencial. Isso significa que o computador lê uma sequência de instruções pela unidade central de processamento (também conhecida como CPU), executa a instrução até o final e move para a próxima instrução. A programação paralela foi criada para otimizar o processamento sequencial de uma grande tarefa computacional, por fazer uso de mais de núcleo de processamento, sendo possível

quebrar uma instrução em instruções menores para serem processadas simultaneamente.

> Existem diversos tipos de computação paralela: nível de bit (BLP, do inglês *Bit Level Parallelism*), nível de instrução (ILP, do inglês *Instruction Level Parallelism*), nível de dados (DLP, do inglês *Data Level Parallelism*) e nível de tarefa (TLP, do inglês *Task Level Parallelism*). Cada um dos tipos opera formas diferentes de lidar com paralelismo.

A técnica de paralelismo é utilizada há vários anos, principalmente na computação de alto desempenho, e recentemente vem se tornando cada vez mais popular devido às limitações físicas que previnem o aumento de frequência de processamento. Essa técnica acabou popularizando o paradigma da programação paralela, que tem como premissa o processamento simultâneo de partes da aplicação, onde haja uma cooperação entre elementos de processamento utilizando fundamentos de comunicação e sincronização.

É bem comum o termo "programação paralela" ser associado, de forma errônea, com "programação concorrente". É importante saber que eles representam princípios distintos. *Concorrência* é a capacidade de lidar com várias instruções, uma por vez, e *paralelismo* é a capacidade de lidar com várias instruções ao mesmo tempo.

Para um exemplo de concorrência na vida real, imagine duas filas de pessoas ("A" e "B") para apenas uma máquina de refrigerantes. Em cada uma das filas "A" e "B", existe uma pessoa pronta para comprar o refrigerante; em outras palavras, se a primeira pessoa da fila "A" estiver bloqueada, prontamente a primeira pessoa da fila "B" ocupará a posição de compra. Outro cenário seria o caso de todas as pessoas da fila "B" estarem bloqueadas, então todos da fila "A"

teriam prioridade. Desta forma, a máquina de refrigerantes lidaria de forma concorrente com ambas as filas.

Para entender o paralelismo, podemos usar o exemplo anterior e imaginar que, em vez de apenas uma máquina de refrigerantes, agora existem duas máquinas de refrigerantes para duas filas. É certo afirmar que há compras de refrigerantes sendo realizadas no mesmo tempo, pois as filas "A" e "B" não precisam dividir a máquina entre si.

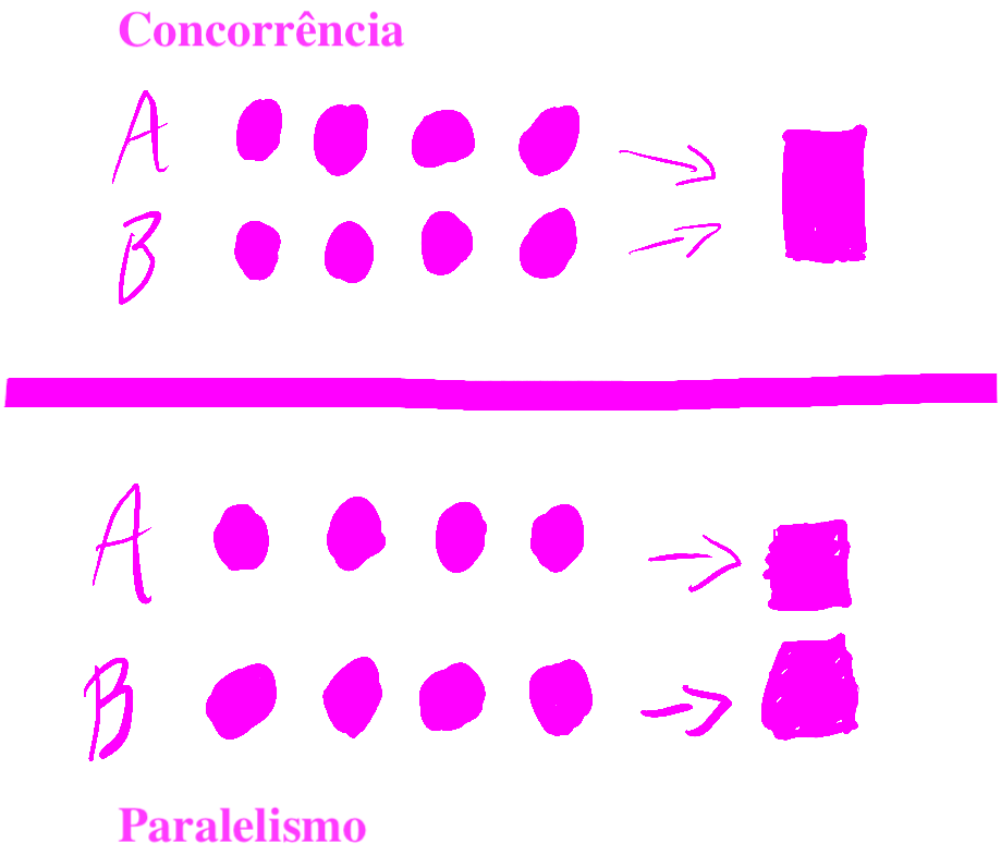


Figura 7.1: Concorrência e paralelismo com o exemplo da máquina de refrigerantes.

De volta ao mundo da programação, podemos afirmar que precisamos de mais uma máquina de refrigerantes, ou seja, mais de um "elemento de processamento" para performar programação paralela. No caso da Web, para ter acesso a essa informação, o navegador provê uma API JavaScript para obter a capacidade de processamento concorrente da máquina. Para acessar esse valor,

basta usar `window.navigator.hardwareConcurrency`. Esse valor pode ser entre "1" e o número de processadores lógicos potencialmente disponíveis para o agente do usuário.

Os computadores modernos têm vários núcleos de processador físico em sua CPU, mas cada núcleo físico geralmente também é capaz de executar mais de uma *thread* por vez usando técnicas avançadas de agendamento. Portanto, uma CPU de quatro núcleos pode oferecer oito núcleos de processador lógico. O número de núcleos de processador lógico pode ser usado para medir o número de encadeamentos que podem ser efetivamente executados de uma só vez sem que eles tenham que mudar de contexto.

No entanto, é importante destacar que o navegador pode optar por relatar um número menor de núcleos lógicos para representar com mais precisão o número de *Workers* que podem ser executados ao mesmo tempo, portanto, não podemos tratar como uma medida absoluta o número de núcleos no sistema do usuário.

Juntando as peças

Vamos juntar as peças de todos os conceitos que vimos neste capítulo e ver de forma prática como funciona. Antes de qualquer coisa, vamos criar a estrutura de nossa aplicação usando `cargo` e a nomearemos como `wasm-thread`:

```
cargo init wasm-thread
```

O objetivo é demonstrar, de forma simples, como múltiplos *Web Workers* podem funcionar como *threads* paralelas, lendo e escrevendo em uma memória compartilhada com operações atômicas no WebAssembly.

Já que não precisamos escrever uma aplicação complexa para isso, que tal escrever uma função que soma atomicamente na memória compartilhada? O código Rust (e eventualmente WebAssembly) é bem singelo, sendo uma função que recebe um valor `u8` e lê na

memória a mesma posição (e também reusa o mesmo tamanho). O código da implementação está a seguir:

src/lib.rs

```
use core::slice::from_raw_parts_mut;

#[no_mangle]
fn soma(valor: u8) -> u8 {
    // Obtém a fatia especificada da memória
    let fatia: &mut [u8] = unsafe { from_raw_parts_mut(1 as *mut u8, 255)
};
    fatia[0] = fatia[0] + valor;
    fatia[0]
}
```

Até então o código não faz nada além de pegar uma fatia específica da memória do WebAssembly. Não tem suporte à memória compartilhada e muito menos suporte a operações atômicas. Tais funcionalidades precisam ser configuradas pelo `cargo` (ou diretamente no `rustc`).

Vamos dizer ao `cargo` que a biblioteca padrão do Rust precisa ser recompilada com tipos atômicos usando a configuração `build-std`. Depois, vamos também adicionar a configuração que permite operações atômicas (`+atomics`), operações de expansão da memória (`+bulk-memory`) e globais mutáveis (`+mutable-globals`).

O `cargo` pode processar essas configurações por argumentos na linha de comando ou em um arquivo de configuração nomeado como `config` no diretório `.cargo`, na raiz do projeto:

.cargo/config

```
[unstable]
build-std = ['std', 'panic_abort']

[build]
target = "wasm32-unknown-unknown"
```

```
rustflags = [
    '-Ctarget-feature=+atomics,+bulk-memory,+mutable-globals'
]
```

É necessário também definir o comportamento da memória como memória compartilhada, do contrário não vamos conseguir compartilhar memória entre *threads*. Para isso, vamos atualizar o arquivo de configuração para que a memória não apenas seja compartilhada, mas também tenha um tamanho máximo e que seja externa ao WASM.

```
[build]
target = "wasm32-unknown-unknown"
rustflags = [
    '-Ctarget-feature=+atomics,+bulk-memory,+mutable-globals',
    '-C', 'link-args=--no-entry --shared-memory --import-memory --max-memory=209715200',
]
```

Nesse código, estamos definindo a memória máxima como `209715200`, que representa `200MiB`, ou seja, 3.200 páginas de memória. Uma vez que sabemos o tamanho máximo da memória, podemos escrever nosso arquivo JavaScript que lida com o WebAssembly, o `app.js`:

app.js

```
const memory = new WebAssembly.Memory({
  initial: 320, // 20 MiB
  maximum: 3200, // 200 MiB
  shared: true
});

const importacoes = { env: { memory } };
```

> Perceba que o objeto da instância não possui os dados da memória, pois definimos que a memória é externa e é compartilhada.

Na parte do JavaScript, necessitamos também incluir os *Web Workers* que serão responsáveis pela comunicação como WebAssembly. Esses *Workers* vão realizar apenas duas operações: a inicialização da instância do WASM e a execução da função "soma".

Outro detalhe importante é que os *Web Workers* serão criados de acordo com o número de processadores lógicos disponíveis e, para isso, usaremos a API `hardwareConcurrency`. Conversamos brevemente sobre essa API anteriormente. Com um comportamento bem simples de apenas leitura de valor, ela retorna o número de processadores lógicos disponíveis para executar *threads* no computador do usuário.

app.js

```
// ...

// Define a lista de Workers
const workersLista = [];

// Número de processadores lógicos disponíveis
const processadores = window.navigator.hardwareConcurrency;
console.log(`processadores: ${processadores}`);

// Cria um Worker para cada processador
for (let i = 0; i < processadores; i++) {
  let worker = new Worker('./worker.js');

  // Adiciona na lista para cada retorno de inicialização
  worker.addEventListener('message', e => {
    if (e.data.operacao === 'INICIALIZAR') {
      workersLista.push(worker);
    }
  }, false);

  // Envia uma mensagem para o worker com a operação de inicialização
  worker.postMessage({ operacao: 'INICIALIZAR', importacoes });
}
```

Com esse código, criamos um *Web Worker* para cada processador lógico disponível e mandamos uma mensagem que contém a operação de inicialização (`INICIALIZAR`) e o objeto que configura as importações.

Vamos então fazer a parte da execução no *app.js*. Quando todos os *Web Workers* forem inicializados, estaremos aptos a inicializar suas respectivas execuções. Podemos esperar pelo número de *Web Workers* inicializados ser igual ao número de processadores e então enviar uma mensagem com a operação de execução com seu respectivo valor. Siga o código:

app.js

```
// ...

// Para cada 100 milissegundos executa a função "executarWorkers"
const intervalo = setInterval(executarWorkers, 100);

function executarWorkers() {
  // Valida se todos os Web Workers terminaram suas respectivas
inicializações
  if (workersLista.length == processadores) {

    // Desabilita a execução da função "executarWorkers" a cada 100ms
    clearInterval(intervalo);

    // Envia uma mensagem com o valor baseado na ordem
    for (let i = 0; i < processadores; i++) {
      const valor = i + 1;
      console.log(`valor para somar: ${valor}`);

      workersLista[i].postMessage({
        operacao: 'EXECUTAR',
        valor
      });
    }
  }
}
```

Esse código conclui a parte do *app.js*. Note que, até então, não possuímos nem a inicialização do WASM nem o *Web Worker*. A novidade é que faremos a inicialização do WebAssembly em cada *worker*. O código a seguir é responsável pela lógica do *Web Worker*:

worker.js

```
// Para evitar instanciar o WebAssembly múltiplas vezes, criamos essa variável
// que auxilia posteriormente quando o Web Worker recebe mensagens com operações de execução
let funcaoSoma;

// Escuta os eventos de mensagens
addEventListener('message', async (e) => {

  // Na operação de inicialização, cria a instância do módulo
  if (e.data.operacao === 'INICIALIZAR') {
    result = await WebAssembly.instantiateStreaming(
      fetch('./target/wasm32-unknown-unknown/release/wasm_thread.wasm'),
      e.data.imports
    );

    // Salva a função soma no escopo externo
    funcaoSoma = result.instance.exports.soma;
  }

  // Na operação de execução, usa a função que foi salva anteriormente no escopo externo
  if (e.data.operacao === 'EXECUTAR') {
    const retorno = funcaoSoma(e.data.valor);
    console.log(retorno);
  }

  // Envia uma mensagem de volta com os dados recebidos anteriormente
  postMessage(e.data);
}, false);
```

Ok! Vamos dar uma olhadinha no navegador? Perceba que a ordem de resposta muda para cada execução, e não apenas isso: os *Web Workers* são criados e executados também em ordem diferentes!

Primeiro exemplo com 8 processadores

```
processadores: 8 app.js:14:9
valor para somar: 1 app.js:36:15
valor para somar: 2 app.js:36:15
valor para somar: 3 app.js:36:15
1 worker.js:13:13
3 worker.js:13:13
valor para somar: 4 app.js:36:15
6 worker.js:13:13
valor para somar: 5 app.js:36:15
10 worker.js:13:13
...
```

Segundo exemplo com 8 processadores

```
processadores: 8 app.js:14:9
valor para somar: 1 app.js:36:15
valor para somar: 2 app.js:36:15
1 worker.js:13:13
valor para somar: 3 app.js:36:15
valor para somar: 4 app.js:36:15
3 worker.js:13:13
6 worker.js:13:13
valor para somar: 5 app.js:36:15
10 worker.js:13:13
...
```

É importante destacar que, diferentemente do exemplo que vimos anteriormente escrito em Rust, este exemplo usa processamento paralelo, porém com o sistema de bloqueio atômico para leitura e escrita do valor na memória.

Podemos concluir que o WebAssembly permite fazer processamento paralelo e concorrente de dados que podem ser compartilhados entre *Web Workers* ou até mesmo outros recursos, já que o WASM fornece todas as ferramentas necessárias para operações atômicas.

No próximo capítulo, vamos ver os tipos de erro do WASM e como evitá-los.

Capítulo 8
Tipos de erro

Neste capítulo, vamos ver os possíveis tipos de erro com WebAssembly e como se apresentam no JavaScript. Também faremos a apresentação dos tipos `Exception`, `Tag`, `CompileError`, `RuntimeError` e `LinkError`, que fazem parte da API do WebAssembly que o JavaScript provê no navegador.

> Lembrete: se você tiver interesse em reproduzir os exemplos de erros deste capítulo, todos os exemplos do livro (incluindo os arquivos binários) estão neste link: https://github.com/raphamorim/livro-webassembly-exemplos.
>
> Basta clonar o repositório usando git ou fazer uso da opção de download do código-fonte que o GitHub provê.

Exemplificaremos diferentes tipos de erro que podem ocorrer, pois temos como objetivo não só aprender a evitar e solucionar, mas também buscar o entendimento dos fundamentos e razões por trás de cada erro.

Os erros do WebAssembly acontecem em diferentes fases do ciclo de vida de um módulo no navegador, que começa no processo de compilação e vai até o tempo de execução do módulo. Neste capítulo, agrupamos cada tipo de erro em sua respectiva fase do ciclo de vida.

8.1 Erro de compilação

Na fase de compilação do WASM no navegador, o erro que pode acontecer é o ***CompileError***. O objeto `WebAssembly.CompileError` indica um erro durante a decodificação ou validação do WebAssembly.

Para reproduzir esse erro, precisamos de um arquivo WebAssembly inválido. Neste caso, vamos criar um arquivo chamado `arquivo-invalido.wasm` e, dentro dele, adicionar um texto escrito "conteudo-do-arquivo-invalido":

arquivo-invalido.wasm

```
conteudo-do-arquivo-invalido
```

Conforme aprendemos anteriormente, esse não é o tipo de conteúdo válido para um arquivo WebAssembly, então, quando executarmos a função `validate` para validar os bytes do arquivo, ela retornará falso.

Veja um exemplo de compilação para um módulo WASM usando o arquivo que escrevemos anteriormente:

```
fetch('arquivo-invalido.wasm').then((response) =>
  response.arrayBuffer()
).then(function(bytes) {
  // Se "valido" for verdadeiro, o conteúdo do arquivo está correto
  const valido = WebAssembly.validate(bytes);

  // Printa no console se é válido ou não
  console.log(`Os _bytes_ são ${valido ? "válidos" : "inválidos"} para
compilar em um módulo WASM`);

  // Tenta compilar os bytes, porém retorna um CompileError
  WebAssembly.compile(bytes);
});
```

Perceba que o erro aparece apenas quando tentamos compilar bytes inválidos. Este é o *CompileError*, um erro de compilação do arquivo binário. Na imagem a seguir, veja um exemplo desse erro no console do navegador:

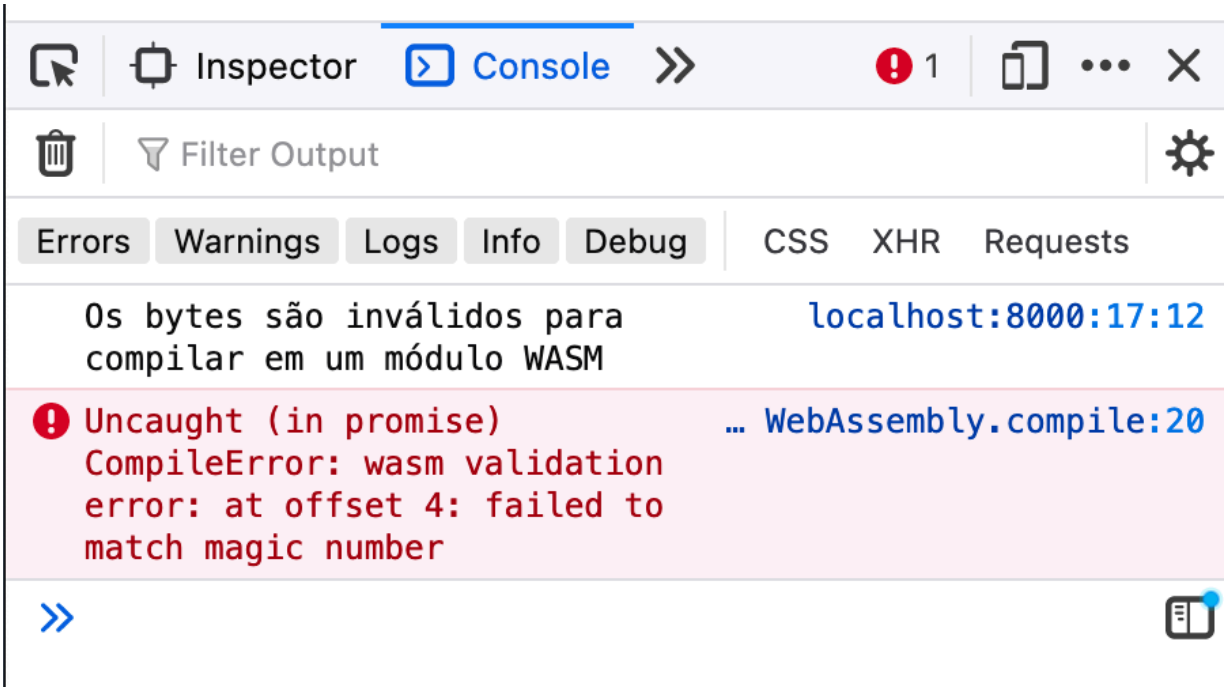


Figura 8.1: Exemplo CompilerError.

8.2 Erro na configuração da instância

Quando criamos uma instância a partir do módulo compilado, podemos ter outro tipo de erro, chamado ***LinkError***. O objeto `WebAssembly.LinkError` indica um erro durante a instância do módulo.

> O *LinkError* também pode acontecer se houver alguma interceptação na função inicial que for definida no módulo compilado. Veremos como as intercepções funcionam na seção *Erros em tempo de execução* deste capítulo.

Para reproduzir o *LinkError*, precisamos primeiro de um arquivo binário válido. Vamos criar nosso arquivo `link.rs`, que também será usado para outros erros além de *LinkError*. Nesse arquivo,

adicionaremos a função `soma_ate_n` com a notação `no_mangle`, que recebe um `Float32` e que retorna também `Float32`:

```
#[no_mangle]
pub extern fn soma_ate_n(n: f32) -> f32 {
    (n * (n + 1.)) / 2.
}
```

Basicamente, a função recebe `n`, soma o valor de `n` com `1.0`, multiplica pelo valor de `n` e divide por dois. Então, se passamos `1.0`, recebemos de volta `1.0`; se for `3.0`, recebemos de volta `6.0`; o retorno muda com o valor de `n`.

Vamos criar um arquivo binário executando o comando a seguir:

```
rustc link.rs --target wasm32-unknown-unknown --crate-type=cdylib
```

Após a execução do comando, temos nosso arquivo WebAssembly. Agora só precisamos do tradicional `index.html` e de adicionar o código para criação da instância no WebAssembly:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Exemplo LinkError</title>
</head>
<body>
  <script>
    // Sucesso
    fetch('link.wasm')
      .then(response => response.arrayBuffer())
      .then(bytes => WebAssembly.compile(bytes))
      .then(mod => WebAssembly.instantiate(mod))
      .then(mod => {
        const { soma_ate_n } = mod.exports;

        console.log('<- recebido: ', soma_ate_n(1)); // 1
        console.log('<- recebido: ', soma_ate_n(2)); // 3
```

```
        console.log('<- recebido: ', soma_ate_n(3)); // 6
      });
  </script>
</body>
</html>
```

Até então, nenhum erro. Todos os `console.log` funcionam perfeitamente, retornando `1`, `3` e `6`. Lembra que vimos no capítulo 3 que `instantiate` também permite passar a configuração da instância no segundo argumento? O parâmetro de configuração é basicamente um objeto que contém os valores a serem importados para a instância recém-criada, como funções ou objetos de memória (`WebAssembly.Memory`).

No objeto de importação, deve haver uma propriedade correspondente para cada importação declarada do módulo compilado, ou então um `WebAssembly.LinkError` é gerado. É justamente na fase de criação da instância que o *LinkError* pode acontecer.

Para reproduzir o erro, vamos declarar uma importação no nosso módulo compilado. Dentro do arquivo `link.rs`, vamos criar a definição de importação usando a palavra-chave `extern` e as funções que queremos usar no nosso código.

Por padrão, a `LLVM` e a `LLD` vão pôr as importações para um módulo nomeado como `env`. Isso significa que, se você compilar este código:

```
extern {
    #[link_name = "console_log"]
    fn log(x: f32) -> f32;
}
```

O compilador vai gerar um arquivo binário que importa a função `console_log` do módulo `env`. Vamos atualizar nossa função `soma_ate_n` para usar a função externa `console_log`, que, dentro do contexto de execução, está nomeada como `log`:

```
extern {
    #[link_name = "console_log"]
    fn log(x: f32) -> f32;
}

#[no_mangle]
pub extern fn soma_ate_n(n: f32) -> f32 {
    unsafe {
        log(n);
    }
    (n * (n + 1.)) / 2.
}
```

Após compilar esse código, vamos escrever o código JavaScript para lidar com a informação de importações. A seguir, o exemplo de como fica o código no JavaScript:

```
const configuracao = {
  env: {
    console_log: (n) => {
      console.log('-> enviado: ', n);
    }
  }
};

fetch('link.wasm')
  .then(response => response.arrayBuffer())
  .then(bytes => WebAssembly.compile(bytes))
  .then(mod => WebAssembly.instantiate(mod, configuracao))
  .then(mod => {
    const { soma_ate_n } = mod.exports;

    console.log('<- recebido: ', soma_ate_n(1)); // 1
    console.log('<- recebido: ', soma_ate_n(2)); // 3
    console.log('<- recebido: ', soma_ate_n(3)); // 6
  });
```

Se juntarmos as peças e executarmos no navegador, tudo funciona perfeitamente, como mostra a imagem a seguir:

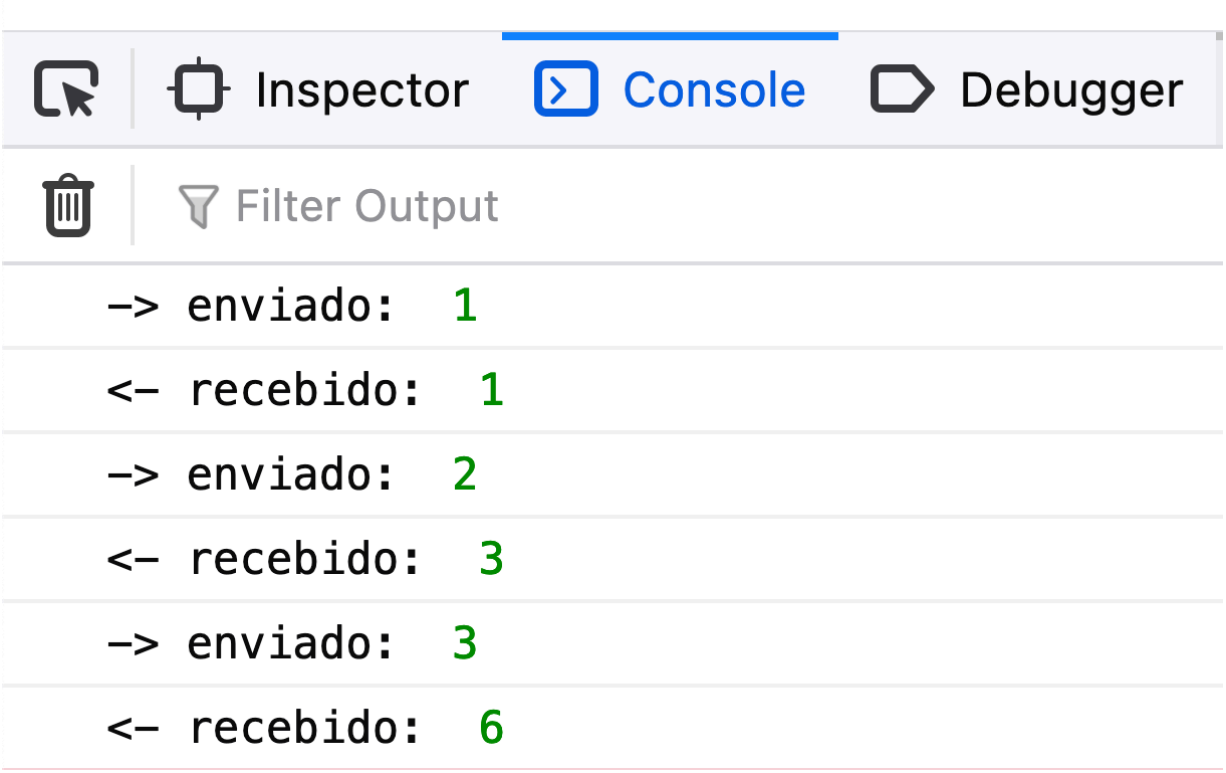


Figura 8.2: Exemplo sucesso com links.

Porém, se mapearmos errado nosso objeto de importação, o `LinkError` vai aparecer. Vamos fazer o `LinkError` aparecer no navegador propositalmente. Basta renomear a `console_log` para qualquer outro nome — vamos usar `log` em vez do nome correto:

```
const configuracao = {
  env: {
    log: (n) => {
      console.log('-> enviado: ', n);
    }
  }
};
```

O erro então aparecerá no console do navegador:

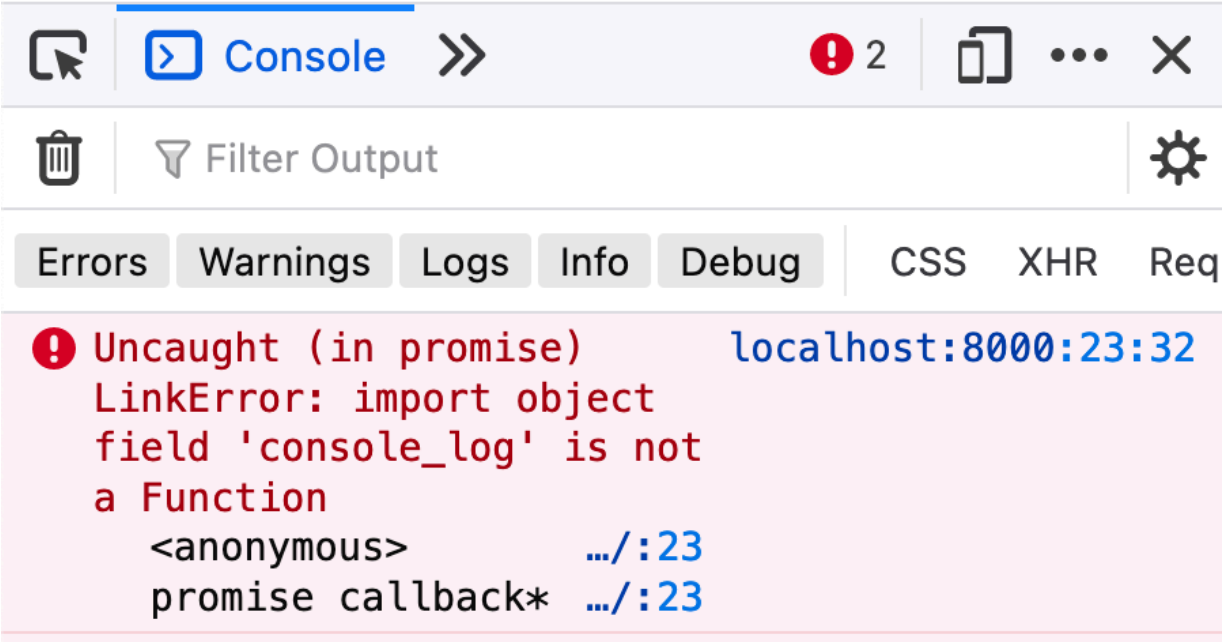


Figura 8.3: Exemplo do LinkError.

Podemos ver que a mensagem do erro especifica ainda que `console_log` não é uma função.

8.3 Erros em tempo de execução

Depois que o módulo WASM foi compilado e instanciado, o WebAssembly pode disparar erros relacionados ao tempo de execução do programa. Esses erros são tipados como *RuntimeError*.

O objeto `WebAssembly.RuntimeError` é o tipo de erro que é gerado sempre que WebAssembly especifica uma interceptação — no âmbito do WASM, esse conceito de interceptação é nomeado como *trap* (armadilha, em português).

A armadilha é executada com base nas instruções dadas ao WebAssembly, que, por sua vez, aborta imediatamente a execução após a execução da armadilha. As armadilhas não podem ser tratadas pelo código WebAssembly, mas são propagadas ao

ambiente externo (exemplo: tempo de execução do JS no navegador), onde normalmente podem ser capturadas.

Alguns exemplos práticos:

- Se tentar o acesso a algo que não esteja dentro dos limites do tamanho da memória atual;
- Se há *"panics"* — *panic* é uma macro usada para construir erros que representam um *bug* que foi detectado no seu programa;
- Se um *parser* recebeu como argumento um formato que não era esperado.

Assim como fizemos anteriormente com *CompileError* e *LinkError*, vamos ver como reproduzir um *RuntimeError*. Para isso, basta forçamos um erro no lado do Rust.

No caso a seguir, a função de nome `funcao_com_erro_em_execucao`, quando for executada, vai disparar o erro, pois realiza a soma entre o valor da variável `numero` e 255. Quando a soma retornar um valor maior que 255, disparará um erro em execução, pois `u8` não suporta valores maiores que 255. Veja o código:

```
// Para gerar bastar executar:
// rustc runtime.rs --target wasm32-unknown-unknown --crate-type=cdylib

#[no_mangle]
pub fn funcao_com_erro_em_execucao(numero: u8) -> u8 {
    numero + 255
}
```

No `index.html`, podemos apenas executar a função e já teremos o `RuntimeError`:

```
WebAssembly.instantiateStreaming(fetch('runtime.wasm'))
  .then(({ instance }) => {
    // Retorna 255
    console.log(instance.exports.funcao_com_erro_em_execucao(0));

    // Retornará um "RuntimeError" pois os
```

```
    // possíveis valores de u8 vão de 0 até 255
    console.log(instance.exports.funcao_com_erro_em_execucao(1));
  });
```

Dito e feito! A seguir, temos o erro de *RuntimeError* no console do navegador:

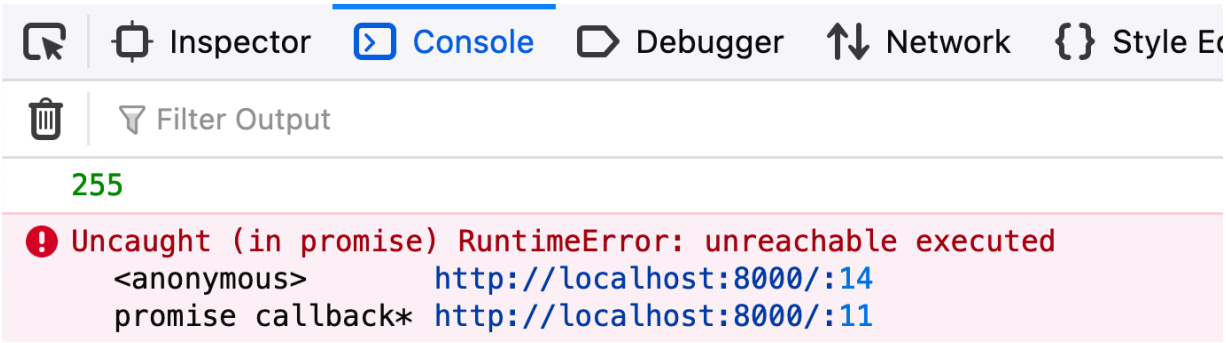


Figura 8.4: Exemplo do RuntimeError.

Perceba que o erro em si não contém muitas informações e, dependendo do tipo de erro que acontecer no tempo de execução, pode ser ainda mais obscuro e difícil depurar o problema posteriormente.

É possível dar mais informações ao erro criando funções auxiliares e escrevendo os respectivos tratamentos de erro no lado do Rust usando `Result`, `map_err`, `unwrap_or_else`, entre outros recursos da linguagem, como no código a seguir:

erro.rs

```
extern {
    #[link_name = "erro"]
    fn erro(x: u8) -> u8;
}

fn funcao_que_falha() -> Result<(), &'static str> {
    Err("Falhou")
}

#[no_mangle]
```

```
pub extern fn executa_funcao_que_falha() {
    match funcao_que_falha() {
        Ok(_) => {},
        Err(..) => {
            unsafe {
                erro(1);
            }
        },
    }
}
```

Nesse exemplo, toda vez que houver um erro, será enviado um número de 0 a 255 que representa um tipo de erro. É possível enviar mais informações do que apenas números, mas veremos, no próximo capítulo, como fazer codificação de dados com WebAssembly.

Outra opção é acionar o lançamento de mensagens quando o `panic` for executado, como o projeto `console_error_panic_hook`, que faz isso de forma automática (https://github.com/rustwasm/console_error_panic_hook).

Lançando exceções no tempo de execução

Acabamos de ver que o *RuntimeError* é o erro que acontece de forma inesperada no tempo de execução. Entretanto, também podemos lançar exceções de erros de forma premeditada. A API do WebAssembly possui dois objetos para trabalhar com exceções de erro: **Exception** e **Tag**.

O objeto `WebAssembly.Exception` representa uma exceção de tempo de execução que pode ser lançada do WASM para o JavaScript, ou lançada do JavaScript para um manipulador de exceção no WebAssembly.

O construtor da *Exception* aceita um `WebAssembly.Tag`, uma matriz de valores e também um objeto de configuração de opções como argumentos. O objeto `WebAssembly.Tag` define um tipo de exceção

que pode ser lançada de, ou para, o código WebAssembly. Quando criamos uma *Exception*, a *Tag* define os tipos de dados e a ordem dos valores transportados pela exceção.

Veja um exemplo do uso de *Exception* e *Tag* no JavaScript no código a seguir:

```
// Cria um objeto Tag e usa para criar a exceção
const tag = new WebAssembly.Tag({
  // Define o tipo dos dois parâmetros como "f32"
  parameters: ["f32", "f32"]
});

// Cria a exceção baseada na Tag criada e usa "85.0" e "23.3" como valores
const exception = new WebAssembly.Exception(tag, [85.0, 23.3]);
```

No código, a *Exception* foi criada a partir da *Tag*; posteriormente, se for preciso acessar os argumentos de uma exceção lançada, é necessário usar a mesma *Tag*. Isso significa que, sem a *Tag* correspondente, a *Exception* criada pode ser capturada e relançada, mas não pode ser inspecionada.

Para tornar o lançamento de exceções rápido, as exceções lançadas do WebAssembly geralmente não incluem o *stack trace*. O código WebAssembly que precisa fornecer um *stack trace* deve chamar uma função JavaScript para criar a exceção configurando a opção `traceStack` no construtor. Veja o exemplo a seguir:

```
// Define o tipo o parâmetro como "i32"
const tag = new WebAssembly.Tag({ parameters: ["i32"] });

// Configura o traceStack para verdadeiro
const opcoes = { traceStack: true };

// Cria a exceção com "stack trace"
const exception = new WebAssembly.Exception(tag, [85], opcoes);
```

Lembra que falei que a exceção pode ser lançada de forma esperada? Vamos fazer um teste de lançamento de exceção do lado

do JavaScript. Primeiro, escrevemos nosso arquivo `excecao.rs`, que basicamente define uma função que vem do objeto de importação e outra função que executa a função importada:

```
// Para compilar execute o comando:
// rustc excecao.rs --target wasm32-unknown-unknown --crate-type=cdylib

extern {
    #[link_name = "lanca_excecao"]
    fn lanca_excecao_js();
}

#[no_mangle]
pub extern fn excecao() {
    unsafe {
        lanca_excecao_js();
    }
}
```

Agora precisamos compilar para o WebAssembly e escrever o código JavaScript que vai criar uma *Tag* que será usada tanto para criar a exceção como para validar a *Tag* da exceção lançada.

```
// Cria a tag com um parâmetro do tipo f32
const tag = new WebAssembly.Tag({ parameters: ['f32'] });

// Define o objeto de configuração de importações e
// especifica uma função chamada lanca_excecao
const configuracao = {
  env: {
    lanca_excecao: () => {
      const exc = new WebAssembly.Exception(
        tag, [85.25], { traceStack: true }
      );
      throw exc;
    }
  }
};
```

```
fetch('excecao.wasm')
  .then(response => response.arrayBuffer())
  .then(bytes => WebAssembly.compile(bytes))
  .then(mod => WebAssembly.instantiate(mod, configuracao))
  .then(mod => { mod.exports.excecao() })
  .catch((e) => {
    console.error(e);

    // Se a exceção for a tag específica, podemos inspecionar
    // usando o método ".is" no erro
    if (e.is(tag)) {
      // Pega o parâmetro de posição 0 da tag
      // Retorna "85.25"
      console.log(`getArg 0 : ${e.getArg(tag, 0)}`);
    }
  });
```

Com o código JavaScript, podemos testar essas mudanças no navegador e, no console, conseguimos ver o formato da exceção:

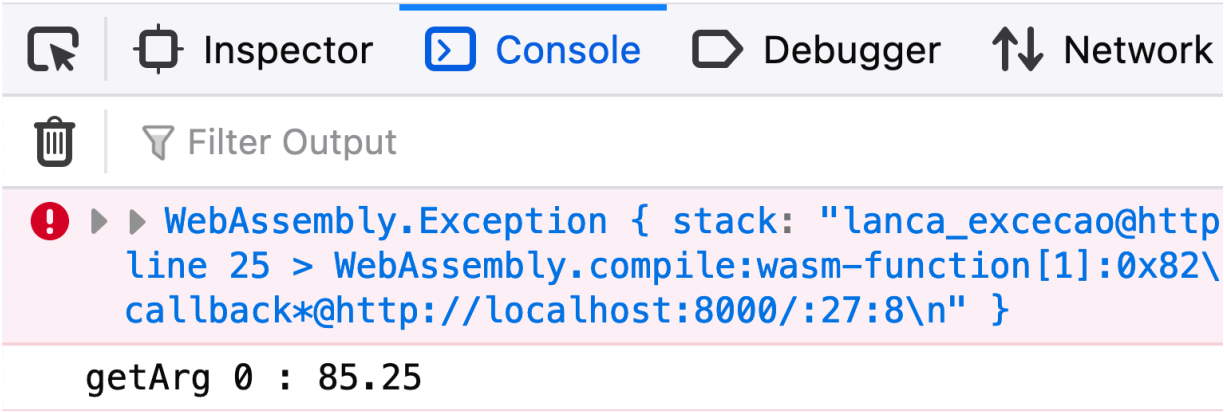


Figura 8.5: Exemplo de Exception com Tag.

É importante destacar mais uma vez que é possível lançar exceções tanto do lado do JavaScript como do lado do WebAssembly. No próximo capítulo, vamos ver um exemplo de lançamento de exceção diretamente no WebAssembly.

Muito bem! Com isso, vimos todos os possíveis tipos de erros que podem surgir no navegador quando estamos trabalhando com

WebAssembly. No próximo capítulo, vamos adentrar no formato de texto do WASM. Veremos como ler, depurar e até mesmo escrever lógica usando o formato `.wat`.

CAPÍTULO 9
Destrinchando o formato de texto

No decorrer dos capítulos anteriores, conversamos brevemente sobre o formato de texto do WebAssembly (`.wat`). No entanto, não chegamos a estudar de fato como funciona sua sintaxe e quais são as suas respectivas características.

Este capítulo é inteiramente dedicado ao funcionamento do formato de texto em termos de sua sintaxe bruta e sua relação com o código binário que a sintaxe representa. De agora em diante, vamos prosseguir escrevendo aplicações WebAssembly na sua representação textual em vez de usar a linguagem Rust.

9.1 WAT: WebAssembly Text Format

O formato de distribuição primário do WebAssembly é um arquivo binário, o .wasm. No entanto, a especificação também descreve uma representação textual chamada *WebAssembly Text Format* (também conhecida como WAT). Esse formato de texto permite que o WASM seja lido e editado por seres humanos.

É uma forma intermediária entre o código binário e os seres humanos, planejada e construída para ser utilizada por editores de texto, navegadores e outras ferramentas de desenvolvimento.

A seguir, uma imagem que exemplifica como o código WAT de uma aplicação WebAssembly é visualizado na aba de depuração do navegador Firefox:

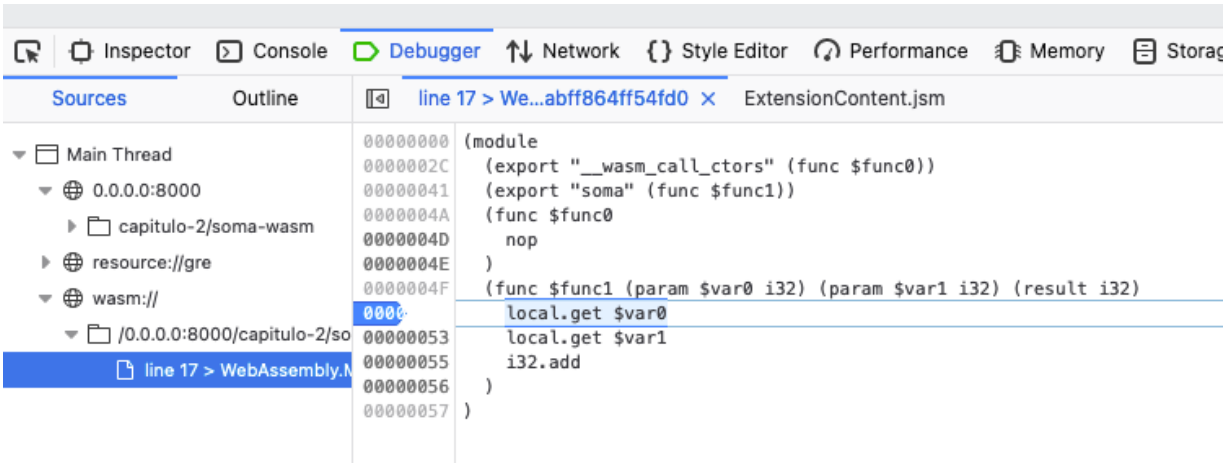


Figura 9.1: Exemplo de formato de texto no navegador.

O navegador Firefox renderiza um arquivo binário descompilado em sua representação WAT. Como todo código binário do WebAssembly pode ser expresso em WAT, é bastante comum o uso do formato de texto para depurar arquivos binários.

Observe que cada instrução do código WAT é envolvida por parênteses e começa com palavras-chave como "module", "export" e "func". Cada uma dessas palavras-chave define o tipo de instrução, e os valores correspondentes complementam a lógica por trás de cada instrução.

Representação por expressões simbólicas

Nos formatos de texto e binário, o conceito fundamental que engloba toda a lógica do código no WebAssembly é um módulo. No formato de texto, um módulo é representado como uma grande expressão simbólica que começa com a instrução `module`. A seguir, veja um exemplo de uma expressão simbólica que inicia um módulo com um comentário.

> Observação: os comentários de várias linhas no WAT são iniciados com a sequência de caracteres `(;` e terminam com `;)`.

```
(module
  ;; código do meu módulo webassembly

  (;
  exemplo de uso do
  comentário de
  múltiplas linhas
  ;)
)
```

A **expressão simbólica** (em inglês, *symbolic expression*), também conhecida como *expressão S*, é uma expressão em uma notação de mesmo nome para dados de lista aninhada (estruturada em árvore). As expressões S foram inventadas e popularizadas pela linguagem de programação Lisp, que as utiliza para o código-fonte da linguagem. Além de ser bem antiga e de leitura simplificada, a expressão S é baseada na representação da estrutura de dados do tipo árvore.

Para ilustrar como funciona a representação, vamos escrever uma expressão S da seguinte expressão matemática (combinação de números, operadores, variáveis livres ou ligadas):

```
X - Y / A
```

Nesse exemplo, podemos resolver as ambiguidades e clarificar a ordem de operações usando parênteses:

```
(X - (Y / A))
```

A representação dessa expressão como uma árvore seria assim:

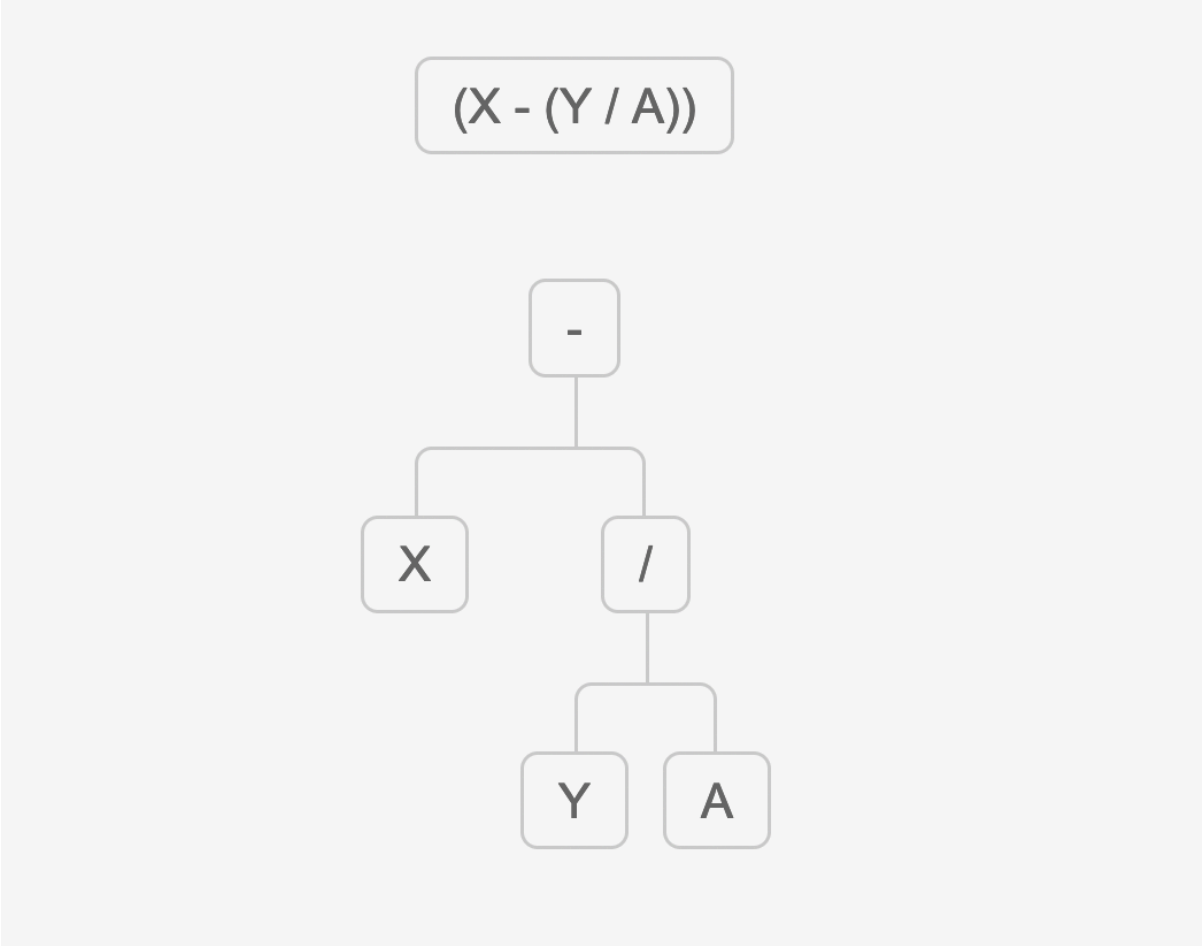


Figura 9.2: Exemplo de expressão em árvore.

Existem diversas formas para podermos transformar essa árvore em uma expressão S. Uma delas é transformar os dados primeiramente em um arquivo JSON e fazer posteriormente algumas pequenas mudanças.

Como o JSON é baseado em chave-valor, vamos usar chaves que identificam o tipo de operação realizada, e os valores serão as respectivas variáveis `X`, `Y` e `A`.

A nomenclatura das chaves devem seguir o padrão definido da expressão S que derivou do Lisp, pois o formato de texto do WebAssembly segue o mesmo padrão. As operações `add`, `sub`, `mult` e `div` derivam do Lisp. Perceba que os termos aritméticos estão abreviados em inglês.

Para indicar em qual posição (direita ou esquerda) o valor está localizado, usaremos os termos recorrentes para fazer essa identificação em uma equação (`lhs` e `rhs` , por exemplo). Veja algumas abreviações de operações em inglês:

- `add` : adição, do inglês, *addition*;
- `sub` : subtração, do inglês, *subtraction*;
- `mult` : multiplicação, do inglês, *multiplication*;
- `div` : divisão, do inglês, *division*;
- `lhs` : lado da mão esquerda, do inglês, *left-hand side*;
- `rhs` : lado da mão direita, do inglês, *right-hand side* .

Existem ainda outras funções aritméticas que veremos no decorrer deste capítulo, mas vamos nos ater às funções básicas por ora.

```
{
  "sub": {
    "lhs": "X",
    "rhs": {
      "div": {
        "lhs": "Y",
        "rhs": "A"
      }
    }
  }
}
```

Esse código nos coloca próximos de obter uma expressão S. A próxima etapa é mudar de colchetes para parênteses, remover as características do arquivo JSON, como aspas, vírgulas e a pontuação de dois pontos, além de remover também as operações `"lhs"` e `"rhs"` . O resultado deve ser igual ao exemplo a seguir:

```
(
  sub (
    X, (
      div (
        Y
        A
```

```
      )
    )
  )
)
```

Por fim, basta nivelar tudo na mesma linha para obtermos, enfim, nossa primeira expressão S:

```
(sub X (div Y A))
```

Expressões no formato de texto do WebAssembly são escritas de forma parecida, porém as operações e instruções que podemos usar são exclusivas da especificação. Se quiséssemos reescrever nosso exemplo para uma expressão válida em WAT, faríamos:

```
(i32.sub (local.get 0) (i32.div_u (local.get 1) (local.get 2)))
```

As instruções que são executadas nesse código são `i32.sub` (subtração) e `i32.div_u` (divisão). É possível notar também que as instruções numéricas estão prefixadas com o tipo de dados em que operam, ou seja, existem diversas instruções de subtração para diferentes tipos de dados, como: `i32.sub`, `i64.sub`, `f32.sub` e `f64.sub`.

> A especificação que faz a lista completa de instruções numéricas disponíveis no WAT pode ser obtida neste link: https://webassembly.github.io/spec/core/text/instructions.html#numeric-instructions. Todas as informações e instruções estão escritas em inglês.

No mesmo exemplo, além das instruções numéricas `i32.sub` e `i32.div_u`, temos também a instrução `local.get`. Essa instrução é usada para recuperar uma variável ou parâmetro definido dentro do escopo de uma função, porém, nesse exemplo de código, não há nenhum valor disponível para ser obtido. Que tal providenciarmos valores no escopo por meio de funções?

Funções

A palavra-chave `func` é responsável por criar funções na representação textual do WebAssembly e pode receber opcionalmente a definição de parâmetros e o seu respectivo resultado.

Se quiséssemos providenciar valores no escopo do exemplo que vimos na seção anterior, poderíamos criar os respectivos parâmetros que representam os valores "X", "Y" e "A". Isso significa que `local.get` usaria o índice da ordem dos parâmetros, como no código a seguir:

```
(func $funcao (param i32) (param i32) (param i32) (result i32)
  (i32.sub
  (local.get 0)
    (i32.div_u (local.get 1) (local.get 2)
  ))
)
```

Note que simplesmente repetimos o valor `i32`, mas também é possível receber identificadores pela conveniência. Os identificadores são todos prefixados com o caractere `$`, como no exemplo a seguir:

```
(func $funcao (param $X i32) (param $Y i32) (param $A i32) (result i32)
  (i32.sub
  (local.get $X)
    (i32.div_u (local.get $Y) (local.get $A)
  ))
)
```

Esse código pode ser visualizado em Rust como:

```
fn funcao(x: i32, y: i32, a: i32) -> i32 {
  x - y / a
}
```

Os tipos de dados nas assinaturas das funções não precisam seguir repetição e podem ser diversos. Veja um exemplo de uma função

que recebe valores do tipo `f32`, `f64` e `i32` em seus respectivos índices:

```
(func (param f32) (param f64) (local i32)
  local.get 0
  local.get 1
  local.get 2)
```

No formato de texto do WebAssembly, você pode configurar de forma opcional uma função que será executada assim que o módulo for instanciado. A instrução `start` de um módulo declara qual função será invocada automaticamente assim que o módulo é instanciado, após a inicialização de tabelas e memórias.

Qualquer função configurada na instrução `start` destina-se a inicializar o estado de um módulo. O módulo e suas exportações não estão acessíveis antes da conclusão dessa inicialização. O código a seguir demonstra o seu uso:

```
(module
  ;; Define uma função sem nenhuma respectiva função
  (func $funcaoVazia)

  ;; Coloca a função "funcaoVazia" como inicializador do módulo
  (start $funcaoVazia)
)
```

As funções também podem definir retornos usando a palavra-chave `result`. Se nenhum retorno for definido pela função, sua configuração será de uma função sem nenhum retorno pelo compilador. O WAT disponibiliza também funções com múltiplos retornos, como no exemplo a seguir:

```
(func $retorna_oito_e_cinco (result i32 i32)
  i32.const 8
  i32.const 5)
```

Para usar essa função, usamos a instrução `call`, que define a execução de uma função. No caso a seguir, `call` executa a função

identificada como `retorna_oito_e_cinco`:

```
(func $subtracao (result i32)
  call $retorna_oito_e_cinco
  i32.sub)
```

Nesse exemplo, a função `subtracao` subtrai os valores fixos da função `retorna_oito_e_cinco`, 8 e 5, e retorna o valor `3` de forma constante para cada execução.

As funções do WebAssembly são bastante parecidas com as de qualquer outra linguagem. Elas possuem uma assinatura que declara nenhum, um ou mais parâmetros e um valor de retorno opcional. O corpo de uma função contém um número de instruções que são executadas sequencialmente.

> Nota: o WebAssembly possui aproximadamente 50 instruções diferentes.

As instruções do WebAssembly gerenciam uma pilha (estrutura de dados) e determinam o valor da pilha sequencialmente. O valor da pilha é inteiramente baseado na sequência de instruções executadas.

Máquina orientada a pilha

Mas, peraí, como assim pilha?

Na Ciência da Computação, uma pilha (*stack*, em inglês) é um tipo abstrato de dado e estrutura de dados baseado no princípio de LIFO (do inglês, *"Last In First Out"*), que assume que o último elemento a ser empilhado é o primeiro que será desempenhado.

A arquitetura de máquina orientada a pilha (*stack machine*, em inglês) segue princípios similares aos da pilha e é uma alternativa a arquiteturas de registradores encontradas nos computadores atuais. O coração do WebAssembly é fundamentalmente baseado na

arquitetura de máquina orientada a pilha para execução de instruções.

A máquina orientada a pilha é geralmente vista em um processador de computador ou uma máquina virtual na qual a interação primária está movendo valores temporários de curta duração de e para uma pilha.

No WebAssembly, a máquina orientada a pilha funciona da seguinte forma: cada instrução gerencia a ordem dos elementos da pilha, podendo empilhar, desempilhar, trocar posições desde que cumpra certas condições, entre outros. De forma prática, podemos ver a manipulação dos valores da pilha em qualquer simples função do WebAssembly; para isso, vamos exemplificar um cenário onde temos a seguinte fórmula:

$$f(x) = 2x^2 - 1$$

Figura 9.3: Representação da fórmula.

A replicação dessa fórmula pode ser feita por uma simples função que recebe o valor de `x` e aplica as operações aritméticas. O exemplo a seguir demonstra o valor da pilha para cada instrução executada:

```
(func $f (param $x i32) (result i32)
  ;; A pilha começa vazia: []

  (local.get $x) ;; pilha: [x]
  (local.get $x) ;; pilha: [x, x]
  (i32.mul)   ;; pilha: [x*x]

  (i32.const 2)  ;; pilha: [2, x*x]
  (i32.mul)   ;; pilha: [2*x*x]
```

```
  (i32.const 1)  ;; pilha: [1, 2*x*x]
  (i32.sub))  ;; pilha: [2*x*x-1]
```

Observe que o valor da pilha muda para cada tipo de instrução executada. Quando executamos `local.get`, estamos empilhando o valor de `$x`. Assim que a operação aritmética de multiplicação (`i32.mul`) é executada, ela compara os dois últimos elementos da pilha e substitui os dois elementos por apenas um: o valor da multiplicação. O mesmo padrão acontece para as seguintes instruções: para cada operação aritmética executada, os últimos dois valores da pilha se tornam um.

O WebAssembly também valida se a função retorna exatamente o que foi definido no(s) retorno(s). Se a função possui `(result f32)`, ao fim da execução da função, o WASM faz a comparação do tipo do valor retornado e, se não estiver de acordo com a definição, lança um erro. Se nenhuma definição de retorno for especificada, a pilha ficará vazia ou se manterá a mesma após a execução da função.

Entender como a máquina orientada a pilha funciona e poder visualizar a movimentação de dados na pilha é extremamente importante para compreender a execução de uma função. Contudo, é importante destacar que a pilha não pode ser depurada ou visualizada se o WebAssembly estiver executando um função externa, uma vez que a execução é realizada externamente.

Importações e exportações

O WAT também provê recursos para definir as funções que serão importadas e/ou exportadas no formato de texto, definidas pelo uso das palavras-chaves `import` e `export`. A seguir, veja exemplos de uso de cada uma, começando com a exportação de funções.

```
;; exporta função "oito" que retorna o valor constante de 8
(func (export "oito") (result i32)
  i32.const 8)

;; exporta função "soma"
```

```
(func $soma (param $lhs f32) (param $rhs f32) (result f32)
  local.get $lhs
  local.get $rhs
  f32.add)
(export "soma" (func $soma))
```

A instrução de importação recebe o nome da função externa e define a assinatura da função no contexto do WebAssembly; assim, é possível usar a função internamente:

```
;; importa a função externa e a nomeia no escopo como "minha_funcao"
(import "funcao_externa" (func $minha_funcao (param f64)))

;; também é possível definir níveis de aninhamento para importação
(import "console" "log" (func $print (param i32)))
```

Globais

Vimos, no decorrer do livro, que o WebAssembly também dá a capacidade de criar instâncias de variáveis globais, acessíveis no JavaScript e compartilhadas em uma ou mais instâncias de `WebAssembly.Module`. Essa funcionalidade é muito útil, pois permite a vinculação dinâmica de vários módulos.

Tal funcionalidade no formato de texto do WebAssembly é definida quando usamos a instrução `global` seguida de seu identificador juntamente com sua definição de importação e tipo. O exemplo a seguir define um global mutável, pela adição do uso da palavra-chave `mut` com o tipo `i32`:

```
(global $valor (import "env" "global") (mut i32))
```

É possível também criar funções para retornar o valor global ou até mesmo incrementar com uma constante de valor 1. No próximo exemplo, perceba que as funções obtêm o valor usando a operação `get` e definem o valor com a operação `set`:

```
(global $valor (import "env" "global") (mut i32))
  (func (export "retorna_valor") (result i32)
```

```
  (global.get $valor))
  (func (export "incrementa_valor")
  (global.set $valor
  ;; Instrução de adição da "$valor" com uma constante de valor "1"
  (i32.add (global.get $valor) (i32.const 1))))
```

A possibilidade de trabalhar com valores globais é extremamente útil, pois permite a vinculação dinâmica de vários módulos. No caso anterior, seria possível ler o valor de `$valor` para diferentes módulos instanciados.

Memória

Se desejarmos usar outros tipos de dados além dos que o WebAssembly fornece (por exemplo, uma *string*), devemos recorrer ao uso da memória.

A memória possui sua própria instrução no formato de texto do WebAssembly: `memory` . A instrução define o número de páginas que a memória possui. O exemplo a seguir define a configuração de memória do módulo WebAssembly com oito páginas (512KiB).

```
(module
  (memory 8))
```

A instrução `memory` também permite configurar a memória como compartilhada pelo uso da palavra-chave `shared` . A memória compartilhada funciona de forma diferente da memória "normal", pois precisa ter um tamanho máximo definido. O próximo exemplo demonstra uma configuração de memória compartilhada com o tamanho inicial de uma página e o máximo de 10 páginas:

```
(module
  ;; 64 KiB até 640 KiB
  (memory 1 10 shared)
```

É permitido também importar a memória criada no JavaScript com o uso da palavra-chave `import` :

```
(import "env" "mem" (memory 1))
```

Com esse código, não apenas importamos a memória, mas também definimos que deve conter pelo menos uma página (64KiB).

O WAT também contém operações para leitura e gravação na memória linear. As operações são divididas pelos tipos de dados, ou seja, para tipos `i32`, temos as operações: `i32.load` (leitura) e `i32.store` (armazenamento).

> Importante: a operação `i32.load` lê 32 bits como `i32`, assim como a `i32.store` salva 32 bits sem conversão.

Para adicionar os dados na memória, basta utilizar as funções aritméticas que vimos anteriormente. No exemplo a seguir, temos tanto a funcionalidade de leitura quanto a de escrita:

```
;; A função exportada "escrita" escreve a constante 85 no endereço 1
(func (export "escrita")
  i32.const 1
  i32.const 85
  i32.store)  ;; salva 85 no primeiro endereço

;; A função exportada "leitura" retorna o valor 85
(func (export "leitura") (result i32)
  i32.const 1
  i32.load)
```

Todas as operações de leitura, como `f32.load`, `i32.load`, `i64.load`, entre outras, retornam o valor na pilha de execução. Isso significa que você obtém o valor após a leitura e pode utilizá-lo no mesmo escopo da função.

Existem outras operações de leitura e escrita além das tradicionais (sem nenhuma conversão). Uma delas é a `i64.store16`, que armazena um inteiro de 16 bits (2 bytes), agrupando um inteiro de 64 bits retirado da pilha.

Outro exemplo de tipos de operação que realizam conversões são as operações de um byte como números inteiros com sinal (`i32.load8_s`) ou sem sinal (`i32.load8_u`). No código a seguir, veja um exemplo de leitura e escrita com inteiro com sinal:

```
;; Armazena um inteiro de 8 bits (23) no endereço 2
(i32.store8 (i32.const 2) (i32.const 23))

;; Lê o inteiro com sinal no endereço 2
(i32.load8_s (i32.const 2))
```

As funcionalidades de leitura e escrita também disponibilizam a opção de configuração do alinhamento dos bits que serão salvos na memória.

```
;; A função exportada "escrita" escreve a constante 85 no endereço 1
(func (export "escrita_alinhada")
  i32.const 1 ;; endereço
  i32.const 85   ;; valor
  i32.store 2 0) ;; 2 = alinhamento de 32-bit/4-byte, 0 = posição

;; A função exportada "leitura" retorna o valor 85
(func (export "leitura_alinhada") (result i32)
  i32.const 1
  i32.load 2 0)
```

O alinhamento de dados é algo que definimos quando sabemos o tipo de dados que estamos salvando na memória. Por exemplo: um caractere (`char`) tem um alinhamento de dados comum de 1 byte, assim como o alinhamento natural de um inteiro em uma máquina de 32-bits é 4 bytes.

É bem comum desenvolvedores de primeira viagem com WASM não especificarem o alinhamento de forma correta e armazenarem dados de forma incorreta, como um alinhamento de `char` em um dado que representa um inteiro ou vice-versa. A falta de alinhamento adequado nos dados salvos pode criar inúmeros problemas, como um ou mais ciclos extras da CPU para leitura dos dados ou até mesmo erros em tempo de execução ou compilação.

Entretanto, normalmente o comportamento padrão dos compiladores é o de adicionar automaticamente bytes extras nas estruturas de dados para que elas sejam compatíveis com os limites dos dados que salvamos. Essa funcionalidade é conhecida como preenchimento (*padding*, em inglês).

Outro recurso do WAT relacionado à memória é a instrução nomeada como `data`, que permite a você escrever uma *string* de bytes juntamente com a posição de seu endereço inicial na memória. A *string* é convertida automaticamente para bytes. A instrução `data` inicializa os dados definidos na posição especificada no momento da instancialização do módulo. A seguir, temos um exemplo com o uso da instrução `data` com uma *string* com o conteúdo "Raphael":

```
;; Definição de uma página na memória externa (64 KiBs)
(import "env" "mem" (memory 1))

;; Dados salvos desde a posição 2
;; "Raphael" será convertido para _bytes_ na memória
(data (i32.const 2) "Raphael")
```

Esse código, quando instancializado, vai inicializar a memória do WebAssembly contendo a *string* "Raphael" a partir da posição 2 no `ArrayBuffer`. Desta forma, podemos definir uma memória inicial com uma *string* para aplicações WebAssembly. A instrução `data` também contém a operação `drop`, que permite descartar a memória em um segmento específico.

Existe outra instrução de armazenamento de dados que funciona usando a instrução `data`: a instrução `data_segment`, que permite armazenar dados na memória de forma bruta em sua inicialização. Na maioria dos casos, é a melhor opção para quando a memória precisa conter valores específicos antes do instanciamento do módulo.

```
(data_segment
  ;; Índice linear da memória
```

```
0
;; Deslocamento: Define como e onde colocar os dados
(init_expr (i32.const 4))
;; Dados
(data 0x55 0x0 0x0 0x0))
```

Esse código inicializa os bytes `4` a `8` na posição "0" da memória com os valores de byte fornecidos, que, se lidos como um inteiro de 32-bits sem sinal, retorna o valor “85”. Detalhe: estamos usando a instrução `init_expr`, que basicamente coloca o valor `i32` no topo da pilha de execução para a próxima instrução.

Tabelas

Às vezes, em um programa, você deseja ter uma variável que aponte para uma função, como um retorno de chamada; você pode fazer coisas como passá-lo para outra função. Pode acontecer também de mudar o valor da variável para uma função diferente.

> Essa atribuição de valores a parâmetros e variáveis é conhecida no C como ponteiros para funções e no JavaScript como *callback*.

As tabelas são basicamente matrizes redimensionáveis que podem ser acessadas por índice do código WebAssembly. Essas matrizes redimensionáveis podem ser compartilhadas entre instâncias e contêm referências a funções. De forma segura, essas referências a funções são mapeadas na memória externa como índices e o WebAssembly não tem a leitura de qual endereço de memória a função está armazenada.

Parte da resolução de muitos problemas de segurança se baseia em esconder os endereços de memória usados pelo programa. Você quer evitar que o código na página Web seja capaz de ver ou manipular endereços de memória diretamente, uma vez que, se houver qualquer código malicioso na página, será possível usar as

informações de onde os dados estão armazenados na memória para tomar proveito de forma maliciosa.

Um cenário que poderia ocorrer se o endereço da memória estivesse exposto é que seria possível mudá-lo para outro endereço, provido pelo código malicioso. Assim, quando fosse executar a função, ela chamaria a outra função com base no novo endereço de memória que foi provido, podendo ter acesso à leitura e manipulação de dados no tempo de execução.

Para criar uma tabela no WAT, é necessário o uso da instrução `table`. A instrução `table` recebe o tamanho inicial da tabela e o tipo de dados que serão armazenados. No exemplo a seguir, é definida uma tabela que contém 3 funções e tem como tipo `funcref`. O tipo `funcref` diz que o tipo dos elementos são referências de funções. A possibilidade alternativa ao `funcref` é o `externref`, que define que os tipos dos elementos são referências externas de funções.

```
(table 3 funcref)
(elem (i32.const 0) $primeira $segunda $terceira)
(func $primeira (result i32)
  i32.const 100)
(func $segunda (result i32)
  i32.const 200)
(func $terceira (result i32)
  i32.const 300)
```

A instrução `elem` é uma lista das funções que devem ser referenciadas pela tabela e a ordem em que devem ser referenciadas. A lista de funções pode conter qualquer conjunto de funções em um módulo, em qualquer ordem, permitindo duplicatas. A instrução `elem` também contém a operação `drop`, que permite limpar a lista de referências.

O valor `(i32.const 0)` dentro da instrução `elem` representa o deslocamento. No exemplo anterior, estamos dizendo que as funções `$primeira`, `$segunda` e `$terceira` serão salvas nos índices de 0 a 2,

respectivamente. Se fosse definido o deslocamento como `i32.const 3`, as funções seriam referenciadas de 3 a 5.

Os índices das tabelas são valores `i32` e dependem do operador `call_indirect` para suas respectivas execuções. É preciso definir também um tipo para a função que será executada dinamicamente usando a instrução `type`, que serve majoritariamente para validação das funções. No código a seguir, vemos um exemplo de uso da tabela que definimos anteriormente:

```
(table 3 funcref)
(elem (i32.const 0) $primeira $segunda $terceira)
(func $primeira (result i32)
  i32.const 100)
(func $segunda (result i32)
  i32.const 200)
(func $terceira (result i32)
  i32.const 300)

(type $retorna_i32 (func (result i32))) ;; o valor precisa ser i32
(func (export "executa_pelo_indice") (param $i i32) (result i32)
  (call_indirect (type $retorna_i32) (local.get $i))
```

A função `executa_pelo_indice` depende do parâmetro `i32` para executar qualquer função. Se a função receber "0" como parâmetro, ela executará a função `$primeira` e retornará o valor "100", e assim respectivamente para as outras funções referenciadas na tabela.

Tabelas podem expandir dinamicamente também com o uso da operação `set`. A instrução `table.set` recebe um valor `i32` com o tipo de valor que será armazenado na tabela (`funcref`, `externref` ou `anyref`). A seguir, um exemplo de uma função exportada que recebe a posição de deslocamento e um valor do tipo `externref` para adicionar na tabela em tempo de execução:

```
(func (export "adiciona_externref") (param $i i32) (param $r externref)
  (table.set $t3 (local.get $i) (local.get $r))
)
```

Essa função permite vincular funções dinamicamente para ter acesso à mesma memória e à mesma tabela. Valores da tabela opcionalmente podem ser copiados para diferentes regiões da tabela com o uso de `table.copy`.

O uso inesperado da tabela, como tentar obter ou executar valores inexistentes, pode levar a erros. Vamos ver como funciona o lançamento de erros no formato de texto do WebAssembly.

Tag e Exception

No último capítulo, vimos os possíveis tipos de erro do WebAssembly e aprendemos como lançar erros em tempo de execução com o uso de *Tag* e *Exception*. Porém, no exemplo dado, definimos a exceção com sua respectiva *Tag* no lado do JavaScript e executamos o lançamento do erro como uma função externa no Rust.

A seguir, vamos fazer quase o mesmo: receber a *Tag* do JavaScript e executar a função de lançamento de erro; porém, desta vez, a função será configurada no WAT e criará a exceção fora do JavaScript. A *tag* no WAT tem sua própria palavra-chave, `Tag`, e recebe o seu identificador juntamente aos respectivos tipos dos parâmetros.

```
(module
  (import "env" "minha_tag" (tag $minhaTag (param f32 f64))
)
```

O lançamento da exceção baseada em sua respectiva *Tag* depende do uso da operação `throw`, que recebe como parâmetro sua respectiva *Tag* (`$minhaTag`) e o valor do erro.

```
;; A função "lancaErro" lança o erro baseado na tag $minhaTag
(func $lancaErro (param $valorDoErro f32)
  local.get $valorDoErro
  throw $minhaTag
)
```

```
;; Define a função externa "funcao_com_erro" que executa $lancaErro
(func (export "funcao_com_erro")
  f32.const 85.5
  call $lancaErro
)
```

No caso anterior, o erro a ser lançado para a *Tag* `$minhaTag` seria nada mais que um valor *float32*, porém, podemos usar outros tipos de dados ou até mesmo criar mensagens de erro que podem ser decodificadas posteriormente — por exemplo, o uso da instrução `data` para salvar uma *byte string* na memória em tempo de execução.

A função `funcao_com_erro` lança erros de forma contínua sem nenhum tipo de validação — geralmente, não é o cenário que esperamos de uma aplicação funcional. Se quiséssemos adicionar um nível de validação para essa função onde recebemos um valor `f32` e apenas lançarmos a exceção se o valor for maior que "85.5", precisamos fazer uso de dois recursos importantíssimos não apenas no WAT, mas também na programação: *instruções numéricas* e *condicionais*.

Na próxima seção, aprenderemos como funcionam os tipos de instruções numéricas e de controle no WAT, sendo certo afirmar que tais instruções são primordiais para um programa lógico. Começaremos com as instruções numéricas.

9.2 Instruções numéricas

Os nomes das instruções numéricas variam para cada tipo de dados do WebAssembly. Os quatro tipos que temos à nossa disposição são `i32`, `i64`, `f32` e `f64` e, por este motivo, existem 4 formas de criar constantes de valores numéricos:

```
;; Todas as instruções de constantes empilham o valor
;; para o topo da pilha de execução
```

```
i32.const 5
i64.const 8
f32.const 8.5
f64.const 5.8
```

> A lista completa de todas as instruções numéricas pode ser acessada na especificação do WebAssembly, disponível em: https://webassembly.github.io/spec/core/exec/numerics.html.

Em cada um dos tipos de valores que o WebAssembly provê, existem suas respectivas operações de adição, arredondamento, verificação, entre outras. As instruções numéricas abrangem:

- Conversão de valores;
- Comparação de valores;
- Operações aritméticas;
- Instruções específicas de valores flutuantes;
- *Bitwise* (operadores lógicos binários).

Essas modalidades de instruções numéricas serão exemplificadas no decorrer desta seção. A primeira modalidade é a conversão de valores, que, por sua vez, possui inúmeros tipos de instruções.

Conversão de valores

Digamos que você deseja converter um valor `i32` para `f32` no WebAssembly. Seria necessário o uso da instrução `convert_i32_s` (i32 com sinal) ou `convert_i32_u` (i32 sem sinal). Veja:

```
i32.const 85    ;; pilha: [85]
f32.convert_i32_s ;; pilha: [85.0]

f32.const 0.01  ;; pilha: [0.01, 85.0]
f32.add         ;; pilha: [85.01]
```

As instruções de conversão são usadas para converter números inteiros em números de ponto flutuante. Existem versões assinadas e não assinadas desta instrução.

Os números não assinados seguem a regra comum de conversão binária para um inteiro. Os possíveis valores de um número inteiro sem sinal de 8 bits são de 0 (`00000000`) a 255 (`11111111`).

Entretanto, quando queremos ter valores numéricos assinados, precisamos recorrer a diferentes métodos de salvar valores binários. Existem métodos que estendem o sistema binário para representar números com sinal, como: sinal-e-magnitude, complemento para um, complemento de dois e "excesso-N".

Não vamos entrar em detalhes de como cada um desses métodos funciona, porém, para facilitar o entendimento de como funciona um método de representação numérica com sinal em binários, vamos ver o caso de representação mais popular do sistema numérico de base 10: o *sinal-e-magnitude*.

O método **sinal-e-magnitude** usa um sinal positivo ou negativo no valor binário, `0` como positivo e `1` como negativo. Em outras palavras, se temos 8 bits, são utilizados 7 bits para representar o valor do número e 1 bit para representar o sinal. Neste caso, o valor pode variar entre diferentes opções, entre -127 (`11111111`) e 127 (`01111111`).

Na operação reversa, de número de ponto flutuante para inteiro, usamos a instrução `trunc` (do inglês, *truncate*), que truncam a parte fracionária do número na hora da conversão. Assim como as instruções de conversão, existem versões com e sem sinal. Veja um código de exemplo:

```
f32.const 58.5  ;; pilha: [58.5]
i32.trunc_f32_s ;; pilha: [58]
```

Para as demais conversões, existem nomenclaturas específicas para cada instrução: a instrução de conversão do valor `i32` para `i64` é

conhecida como `extend` e a operação reversa, de `i64` para `i32`, é conhecida como `wrap`. Os valores de ponto flutuante possuem outras nomenclaturas também.

Veja um exemplo de extensão (`extend`):

```
;; coloca um i32 no topo da pilha
i32.const 10
;; extensão com sinal de i32 para i64
i64.extend_i32_s
```

No caso de números de ponto flutuante de 32 bits para 64 bits, a instrução é `promote` (promoção), e o contrário, de `f64` para `f32`, é `demote` (rebaixamento). Existe também uma instrução de conversão "especial" chamada `reinterpret`, usada para reinterpretar os bits de um número como um tipo diferente.

As instruções numéricas de conversões são muito úteis, especialmente quando precisamos comparar valores com tipos diferentes.

Comparação de valores

A comparação de valores é realizada por instruções que retornam `0` ou `1`. Se a lógica da comparação for verdadeira, retornará `1`; do contrário, retornará `0`.

Existem seis tipos de comparações numéricas no WAT:

- igualdade (`eq`);
- desigualdade (`ne`);
- "maior do que" (`gt`);
- "menor do que" (`lt`);
- "maior ou igual a" (`ge`);
- "menor ou igual a" (`le`).

Veja a seguir alguns exemplos de uso de algumas dessas instruções de comparação:

```
(func (result i32)
  f32.const 85.5  ;; pilha: [85.5]
  f32.const 58.5  ;; pilha: [58.5, 85.5]

  ;; retorna 1, pois 85.5 é maior que 58.5
  f32.gt          ;; pilha: [1]
)

(func (result i32)
  i32.const 12  ;; pilha: [12]
  i32.const 34  ;; pilha: [34, 12]

  ;; retorna 0, pois 12 é diferente de 34
  i32.eq          ;; pilha: [0]
)
```

Assim como as instruções de conversão, a maioria das instruções de comparação possui versões com e sem sinal. Outra nota importante é que há uma instrução de comparação chamada `eqz`, mas seu uso é restrito para números inteiros e seu objetivo é comparar se o número é igual a zero.

Operações aritméticas

Vimos quase todas as instruções numéricas de tipo aritmético no início deste capítulo: adição (`add`), subtração (`sub`), divisão (`div`), multiplicação (`mul`). A peça restante é a instrução para obter o resto: `rem` (do inglês, *remainder*) e funciona de forma similar ao operador `%` em muitas linguagens de programação. O código a seguir demonstra o uso do `rem` :

```
i32.const 8 ;; pilha: [8]
i32.const 5 ;; pilha: [5, 8]

;; Calcula o resto da divisão entre 8 e 5
i32.rem_u   ;; pilha: [3]
```

As instruções `rem` estão disponíveis somente para os valores de tipo inteiro (`i32` e `i64`). As instruções aritméticas possuem versões com

e sem sinal apenas para as operações de divisão e resto.

Instruções específicas de valores flutuantes

Existe outra categoria de instruções numéricas conhecida como **instruções específicas de ponto flutuante**. Nesta categoria de instruções, não existem versões diferentes para cada instrução baseada em com ou sem sinal, pelo fato de estarmos lidando com números de ponto flutuante e não com inteiros.

Há 10 instruções específicas de ponto flutuante, cada uma com sua respectiva funcionalidade:

- `floor` : retorna o valor arredondado para o próximo menor valor inteiro;
- `ceil` : retorna o valor arredondado para o próximo maior valor inteiro;
- `nearest` : retorna o valor arredondado para o próximo valor inteiro;
- `max` : retorna o maior número entre dois valores de pontos flutuantes;
- `min` : retorna o menor número entre dois valores de pontos flutuantes;
- `neg` : aplica negação ao número — por exemplo, `85.0` torna-se `-85.0` ;
- `trunc` : trunca o número de ponto flutuante para obter o valor sem a sua parte fracional. A instrução `trunc` (`f32.trunc` e `f64.trunc`) se diferencia da `floor` quando é usada em números negativos, pois arredonda para o maior número;
- `sqrt` : calcula a raiz quadrada de um valor;
- `abs` : retorna o valor absoluto de um número;
- `copysign` : copia o sinal do segundo número para o primeiro número.

A seguir, veja diferentes usos de algumas das instruções listadas anteriormente:

```
f32.const 85.0   ;; pilha: [85.0]
f32.const -5.80  ;; pilha: [-5.80, 85.0]
f32.copysign     ;; pilha: [-85.0]
f32.const 85.0   ;; pilha: [85.0, -85.0]
f32.max          ;; pilha: [85.0]
f32.neg          ;; pilha: [-85.0]
f32.const 5.5    ;; pilha: [5.5, -85.0]
f32.add          ;; pilha: [-79.5]
f32.nearest      ;; pilha: [-80]
```

Perceba que cada operação manipula a máquina orientada a pilha de forma diferente; por exemplo, certas instruções esperam dois valores da pilha (`max` e `min`) para transformar em um único valor, assim como o caso da maioria das instruções aritméticas.

Para finalizar esta seção sobre instruções numéricas, temos as instruções *bitwise*.

Bitwise (operadores lógicos binários)

Atualmente, a maioria das linguagens populares de programação possuem suporte para operadores *bitwise*. Esses operadores são usados quando precisamos realizar operações em nível de bits com números inteiros, em outras palavras, trabalhar com sistema binário (ou de base 2).

Dado que a representação binária é dependente de números (especificamente falando dos dígitos binários, `0` e `1`), é classificada dentro da especificação do WebAssembly como uma instrução numérica.

Os populares operadores *bitwise* que são comuns em diversas linguagens de programação são: `and` , `or` , `xor` e `not` , juntamente com as operações *bitshift*. As operações *bitshift* são responsáveis pelo deslocamento e trocas de bits. O WebAssembly possui dois tipos de instruções *bitshift*; o `shl` , que move para esquerda (similar ao `<<` no JavaScript), e o `shr` , que move para direita (é similar aos

sinais `>>` e `>>>` no JavaScript, pois o WASM também provê a opção de uso em números com ou sem sinal).

O WebAssembly também possui as instruções `and`, `or`, `xor`; no entanto, não há suporte ao `not` atualmente e existe uma forte opinião negativa sobre sua implementação na especificação da linguagem. A instrução `and` (similar ao `&` no JavaScript) atua bit a bit e retorna um `1` em cada posição de bit onde os bits correspondentes em ambos são 1(s).

```
i32.const 5   ;; pilha: [00000101]
i32.const 12  ;; pilha: [00001100, 00000101]

i32.and       ;; pilha: [00000100] -> 4
```

A instrução `or` (similar ao `|` no JavaScript) retorna `1` em cada posição de bit para a qual os bits correspondentes de um dos valores são 1(s), mas não de ambos.

```
i32.const 5   ;; pilha: [00000101]
i32.const 3   ;; pilha: [00000011, 00000101]

i32.or        ;; pilha: [00000111] -> 7
```

A última instrução dos clássicos operadores *bitwise* implementados no WebAssembly é a instrução `xor`, que funciona como o operador `^` no JavaScript. Sua instrução retorna `1` em cada posição de bit para a qual os bits correspondentes de um dos valores são 1(s), mas não de ambos.

```
i32.const 7   ;; pilha: [00000111]
i32.const 5   ;; pilha: [00000101, 00000111]

i32.xor     ;; pilha: [00000010] -> 2
```

Existem outras instruções especiais *bitwise* no formato de texto do WebAssembly, como rotação de bits (deslocamento circular) e contadores. As instruções de contadores são diversas e podem obter tanto os zeros de uma estrutura binária quanto os números 1.

Aprendemos nesta seção que podemos trabalhar tanto com comparação de valores, conversão de tipos, especificamente com números de ponto flutuante ou com representações binárias.

Sabendo que você pode verificar se dois números inteiros são iguais, como realizar o controle das próximas instruções baseadas no resultado da comparação? É possível realizar o controle usando instruções *bitwise* ou até mesmo instruções de comparação que retornam 0 ou 1, como muitos registradores funcionam hoje em dia.

Todavia, isso acaba tornando o código WAT ilegível ou de difícil interpretação para qualquer ser humano, especialmente quando o código cresce eventualmente. As instruções de controle do WebAssembly foram criadas para resolver esse tipo de problema.

9.3 Instruções de controle

As instruções de controle do WebAssembly servem para configurar como o fluxo de controle da aplicação vai funcionar. O fluxo de controle é a ordem na qual declarações individuais, instruções ou chamadas de função de um programa imperativo são executadas ou avaliadas. A ênfase no fluxo de controle explícito distingue uma linguagem de programação imperativa de uma linguagem de programação declarativa.

Já sabemos que o formato de texto do WebAssembly é inspirado no Lisp, que, por sua vez, suporta uma mistura de programação procedural e funcional. Ou seja, temos um pouco do paradigma imperativo e declarativo na linguagem. Até então, vimos a grande maioria dos exemplos deste capítulo com fundamentos da programação funcional.

A programação funcional dita que os programas são construídos aplicando e compondo funções. É um paradigma de programação declarativa no qual as definições de funções são árvores de

expressões que mapeiam valores para outros valores, em vez de uma sequência de declarações imperativas que atualizam o estado de execução do programa. No caso do WAT, é fundamentado pelas expressões simbólicas (expressão S).

Porém, também temos suporte a recursos clássicos da programação procedural, que é um tipo de programação imperativa no qual o programa é construído a partir de um ou mais procedimentos (também chamados de sub-rotinas ou funções). Qualquer procedimento pode ser executado a qualquer momento durante a execução de um programa, inclusive por outros procedimentos ou por si mesmo.

Os processadores computacionais fornecem suporte da comunicação do hardware na programação procedural por meio de um registrador de pilha e instruções para chamar e retornar os valores de tais procedimentos. Podemos ver esse conceito aplicado na forma como o formato de texto do WebAssembly depende do uso de instruções para trabalhar com a máquina orientada a pilha.

> A título de curiosidade, existem outros tipos de processadores computacionais que trabalham de forma diferente do padrão atual (registradores orientados a pilha fundamentados em instruções), como as máquinas Lisp ou até processadores Java, que implementam a máquina virtual do Java no hardware; porém, nenhum outro obteve sucesso comercial igual ao modelo padrão de processadores atuais.

As linguagens procedurais geralmente oferecem recursos como palavras reservadas que atuam em blocos, como `if`, `while` e `for`, para implementar o fluxo de controle. Esses recursos do paradigma procedural existem tanto no Rust, JavaScript e no formato de texto do WebAssembly. No WAT, vimos alguns destes recursos de controle, como `call` (chama uma função) e `return` (retorna um valor à função).

Condicionais

Existem outras instruções de controle onde podemos criar blocos específicos, aplicar laços de repetição ou especificar condicionais para valores específicos. Por exemplo, você se lembra que algumas instruções de comparação retornam o valor 0 ou 1 baseado no resultado da operação? Pois bem, podemos criar condicionalismos com as palavras-chaves `if`, `else` e `then`.

A seguir, a função `compara_igual` salva uma *byte string* na memória com base nos valores dados à função:

```
(func (export "compara_igual") (param $numeroA f32) (param $numeroB f32)
  local.get $numeroA
  local.get $numeroB

  ;; Compara se $numeroA é igual $numeroB
  f32.eq
  (if
  (then
  ;; Os valores da memória da posição 10 a 20 são dedicados a esta função

  ;; Dados salvos desde a posição 10
  ;; "Igual" será convertido para _bytes_ na memória
  (data (i32.const 10) "Igual")
  )
  (else
  ;; Dados salvos desde a posição 10
  ;; "Diferente" será convertido para _bytes_ na memória
  (data (i32.const 10) "Diferente")
  )
  )
)
```

No caso anterior, estamos comparando valores e temos dois caminhos possíveis: *igual* ou *diferente*. Todavia, há casos em que apenas queremos obter o valor baseado em uma comparação específica. Um exemplo seria quando temos dois números de ponto flutuante de 32 bits e queremos ficar com o menor valor entre eles.

Seria um pouco fora da curva escrever um `if` e `else` só para retornar o menor valor. Para esses casos, existe a palavra-chave `select`.

A palavra-chave `select` basicamente seleciona o valor com base no resultado da condição dada ou número inteiro, que precisa ser necessariamente 0 ou 1. Funciona de forma parecida com o operador ternário, mas sem avaliação de curto-circuito (também conhecida como *avaliação mínima*), que tem como regra ignorar o processamento do valor do segundo argumento. No caso do WAT, ambos os argumentos são processados independentemente. Veja a seguir como seria o uso do `select` na função `seleciona_menor`:

```
(func (export "seleciona_menor") (param $numeroA f32) (param $numeroB f32)
(result f32)
  local.get $numeroA
  local.get $numeroB

  ;; Compara se $numeroA é maior ou igual ao $numeroB
  ;; Retorna 0 se falso
  ;; Retorna 1 se verdadeiro
  f32.gte

  ;; Seleciona 0 ($numeroA) se $numeroA é menor ao $numeroB
  ;; Seleciona 1 ($numeroB) se $numeroA é maior ao $numeroB
  select
)
```

Nesse exemplo, usamos o benefício lógico da operação reversa à operação *menor ou igual* (`lte`) para retornar justamente o oposto, pois se usássemos `lte`, ela retornaria 1 para o verdadeiro, mas selecionaria o segundo número, uma vez que a instrução `select` usa 0 e 1 quase como um índice de qual valor obterá na pilha.

Tanto a instrução `select` quanto `if` e `else` são bastante usadas junto aos blocos. Os blocos consistem em uma ou mais declarações de instruções e são criados com o uso da palavra-chave `block`.

Blocos

A palavra-chave `block` recebe um identificador que pode ser usado depois em outra instrução de controle chamada `br` para colocar um final no bloco identificado.

Digamos que você tenha uma função que salva uma respectiva idade na memória apenas se a idade cumprir com certas expectativas. O controle por bloco cai como uma luva e permite que você termine a execução do bloco se as expectativas não forem cumpridas, tornando o resto do código inalcançável, já que não é processado.

No exemplo a seguir, temos uma função `$validaIdade` que verifica se a idade é maior ou igual a `18` ; do contrário, encerra a execução do bloco. Se a idade for válida, será executada uma função externa.

```
(func $validaIdade (param $idade i32)
  (block $meuBloco
  local.get $idade
  i32.const 18
  i32.gte_u
  ;; Finaliza a execução do bloco "meuBloco"
  (if (then br $meuBloco))

  ;; Esta instrução é inalcançável se a idade for maior que 18
  call $funcaoExterna
  )
)
```

Código inalcançável

O WebAssembly também provê a instrução que representa a lógica de código inalcançável, nomeada como `unreachable` . O código inalcançável é essencialmente uma linha ou blocos de código que nunca serão executados em tempo de execução.

Se por algum motivo o código definido como inalcançável for executado em tempo de execução, isso vai criar uma armadilha incondicional no WebAssembly. No exemplo dado anteriormente, se adicionarmos a palavra-chave `unreachable` antes da instrução `call`, estaremos criando um erro em tempo de execução.

```
(if (then br $meuBloco))

;; Lançará uma armadilha se a idade for maior que 18
unreachable

call $funcaoExterna
```

Sabemos que a armadilha é extremamente poderosa, pois é um tipo de erro incapaz de ser tratado pelo código WebAssembly, além do fato de que o lançamento da armadilha resulta no fim da execução da instância.

Drop

Existe uma instrução em especial que permite gerenciar os valores da pilha para cada vez que é executada. A instrução `drop` desempenha o último valor da pilha e o descarta. A seguir, veja um exemplo de uso do `drop` em uma função que possui duas constantes de tipos diferentes; entretanto, é necessário o retorno como `i32`:

```
(func $retornaInteiro (result i32)
  i32.const 8 ;; pilha: [8]
  f32. 5.0    ;; pilha: [5.0, 8]

  drop        ;; pilha: [8]
  return
)
```

A instrução `drop` removeu o último valor (`5.0`) e permite à palavra-chave `return` obter o número inteiro (`8`) para o retorno da função. Se desejássemos continuar descartando valores da pilha, precisaríamos escrever a instrução `drop` repetidamente. Para

resolver esse tipo de problema, existe uma instrução de controle que cria laços de repetição chamada `loop`.

Loop

Os laços de repetição criados com `loop` funcionam de forma similar aos blocos: tem um identificador e faz uso da palavra-chave `br`. A diferença das instruções `block` e `loop` é a forma como ambas trabalham com o uso do `br`. Nos blocos, a execução da palavra-chave `br` terminará a execução do bloco, enquanto no laço de repetição moverá a execução para o início. Isso significa que a repetição no WebAssembly não ocorre de forma automática, é necessário que você coloque `br` (ou sua variante como `br_if`) para mantê-la.

Em um cenário onde desejamos criar algo similar a `for`, que incrementa para cada repetição o valor `1` em uma variável de número inteiro que contém `1` até alcançar `8`, precisamos fazer uso da palavra-chave `loop`, assim configuramos o identificador e a sequência de instruções que serão chamadas para cada repetição. O exemplo a seguir reflete o cenário descrito:

```
(func
  ;; Cria uma variável e inicializa no escopo
  (local $i i32)
  ;; Define o valor da variável como 1
  (local.set $i (i32.const 1))

  ;; Cria o laço de repetição nomeado como $repeticao
  (loop $repeticao

  ;; Obtém o valor $i e incrementa com 1
  local.get $i
  i32.const 1
  i32.add
  local.set $i

  ;; Compara se o valor de $i é menor que 9
```

```
  local.get $i
  i32.const 9
  i32.lt_s
  ;; br_if continuará com o laço $repeticao
  ;; se o último valor da pilha for igual a 1
  br_if $repeticao
  )

  ;; retorna o valor 8
  local.get $i
)
```

Entre as instruções executadas dentro do laço, poderíamos executar funções externas, aplicar operações matemáticas, simplesmente iterar diferentes atualizações na memória, entre tantas outras possibilidades. A instrução de controle `loop` é amplamente usada e serve a múltiplos propósitos.

As instruções `loop`, `if`, `else` e `block` também podem fazer uso da palavra-chave `end`, que basicamente dita o término da instrução. É menos usada quando escrevemos WAT puramente baseado em expressões simbólicas, mas, quando usamos o navegador para depurar código de texto do WebAssembly, frequentemente sua representação é escrita com o uso de `end`. Veja o exemplo:

```
(func $verificaSaldoParaSaque (param $saldo i32) (param $valor i32)
  local.get $saldo  ;; pilha: [85.803]
  local.get $valor  ;; pilha: [6.23, 85.803]

  f32.gt          ;; pilha: [1]
  if
  call $realizaSaque
  end
)
```

Nenhuma operação

Vimos quase todas as instruções de controle de fluxo. A última instrução é a instrução de controle `nop` (do inglês, *no-operation*),

que faz literalmente nada e é frequentemente usada para tornar explícito que não há nenhuma operação.

```
(func $funcaoQueNaoFazNada
  ;; A instrução "nop" não realiza nenhuma operação
  nop
)
```

A instrução `nop` é usada quando desejamos declarar que não há nenhuma operação na função. Não há nenhum tipo de benefício adicional do uso da instrução como otimização de compilação, muito menos na redução do tamanho do arquivo, até porque a instrução `nop` é representada por alguns bits no formato binário.

Com isso, concluímos a visualização de uso de todas as instruções de fluxo de controle. Por falar em conclusão, também chegamos ao final deste capítulo. Nestas seções, não só entendemos como funciona a representação textual, mas também como podemos escrever módulos WASM baseados no formato WAT. Isso nos dá uma liberdade enorme, pois não estamos mais restritos à dependência de nenhuma outra linguagem senão o próprio WebAssembly.

Outro benefício obtido no final deste capítulo é que simplesmente podemos visualizar código WAT e depurar código linha por linha. O sentimento de depurar algo desconhecido já não é algo familiar após aprender a leitura de representações por expressões simbólicas, máquina orientada a pilha e as instruções do WASM. Daqui até o final do livro, vamos escrever todos os módulos usando a representação textual e, para isso, precisamos aprender como compilar WAT para WASM.

Este é um dos pontos que vamos visualizar no próximo capítulo, pois estudaremos o formato binário do WebAssembly juntamente do *WABT* e do *Binaryen*. Os pacotes WABT e Binaryen proveem diversas ferramentas que lidam com a representação textual e binária, permitindo até mesmo criar módulos diretamente da representação textual.

Capítulo 10
Estruturas binárias, WABT e Binaryen

A partir de agora escreveremos módulos WASM usando sua representação textual, o `.wat`. Todos os exemplos deste capítulo (incluindo os arquivos binários) estão disponíveis neste link: https://github.com/raphamorim/livro-webassembly-exemplos.

Anteriormente, aprendemos como o WAT funciona tanto em sua semântica quanto no nível da máquina virtual do WebAssembly. Entender como a representação textual do WASM funciona é extremamente importante, pois permite escrever módulos usando o formato de texto, depuração do arquivo binário e até mesmo a construção do seu próprio compilador WebAssembly.

Até então, não chegamos de fato a transformar o formato de texto diretamente para o formato binário, operação que veremos neste capítulo na seção sobre ferramentas binárias. Este capítulo também detalhará melhor sobre como funciona a estrutura e outras características de um arquivo binário.

Além da conversão de `.wat` para `.wasm`, vamos aprender sobre descompilação, estrutura, seções e otimização de arquivos binários. Também veremos a transformação de binários para arquivos de texto, pseudocódigo e linguagens de programação como JavaScript e C.

10.1 WebAssembly Binary Toolkit

Atualmente, existe um pacote oficial para o manuseio de formatos binários ou textuais do WebAssembly, conhecido como WABT (do inglês,: *WebAssembly Binary Toolkit*). O WABT fornece ferramentas

para conversão de formatos `.wat` para `.wasm` (`wat2wasm`) e vice-versa (`wasm2wat`).

O pacote de ferramentas binárias também provê outras funcionalidades, como uma ferramenta para converter um arquivo `.wasm` para um código-fonte em C com seu respectivo arquivo de cabeçalho (`wasm2c`).

Algumas das ferramentas binárias que o pacote provê são:

- `wat2wasm` : traduz do formato de texto do WebAssembly para o formato binário do WebAssembly;
- `wasm2wat` : faz a operação inversa do `wat2wasm` , ou seja, traduz do formato binário para o formato de texto;
- `wasm-objdump` : imprime informações dado um arquivo binário. Funciona de forma semelhante ao comando `objdump` do Linux;
- `wasm2c` : converte um arquivo binário WebAssembly em uma fonte e cabeçalho da linguagem de programação C;
- `wasm-decompile` : faz a descompilação de um binário WASM em uma sintaxe legível que é semelhante à linguagem de programação C;
- `wat-desugar` : analisa a forma de texto `.wat` conforme suportado pelo interpretador de especificações (por exemplo, a validação da sintaxe das expressões simbólicas usadas) e imprime o formato plano.

Você pode ver a lista completa das ferramentas no repositório oficial do GitHub, disponível em: https://github.com/WebAssembly/wabt.

A maioria das ferramentas do WABT são usadas para o desenvolvimento de compiladores ou sistemas que desejam manipular arquivos WebAssembly. Um exemplo é o pacote `wast2json` , que converte arquivos `.wast` (versão histórica do `.wat` que é usada exclusivamente em testes internos da especificação) para o formato JSON juntamente do arquivo binário.

Ao contrário do interpretador oficial da especificação WebAssembly, que é escrito para ser o mais simples e declarativo possível, o WABT

é projetado para facilitar a integração em outros sistemas.

O WABT pode ser usado diretamente do seu código C ou instalando o pacote pronto para uso na linha de comando. Como vamos prover os arquivos WAT escritos "na mão", além do fato de que queremos evitar o uso de qualquer outra linguagem de programação, utilizaremos as ferramentas do WABT no nível da linha de comando.

Para instalar os pacotes do WABT como aplicação de linha de comando, siga a respectiva instrução com base no seu sistema operacional:

- OS X usando *Homebrew* como gerenciador de pacotes: `brew install wabt`
- Debian: `apt-get install wabt`
- Ubuntu: `apt-get install wabt`
- Arch Linux: `pacman -S wabt`
- Kali Linux: `apt-get install wabt`
- Windows (*WSL2*): `apt-get update && apt-get install wabt`

> No caso do Windows, dado o nível de permissão provido ao gerenciador de pacotes, talvez seja necessário executar ambos os comandos com permissão do administrador do sistema (`sudo`).

Existem outras formas de instalação do WABT como aplicação de linha de comando, tais como clonar o repositório do GitHub e fazer o processo de compilação e instalação manualmente. Outras opções seriam usar soluções de pacotes prontos como do NPM, WAPM (destaque para a opção `wasmer/wabt`) ou até mesmo imagens Docker que possuem WABT instalado.

Assim que a instalação das ferramentas do WABT for concluída, vamos colocá-las em prática.

Representação textual para formato binário

Como dito anteriormente, a conversão do código WAT para o formato binário usando as ferramentas do WABT é baseada no uso do `wat2wasm`. O uso da ferramenta em si é bem simples, ela essencialmente recebe um arquivo de texto como argumento e permite adicionar parâmetros customizados.

Para testar o comando de forma rápida, que tal escrevermos o módulo WebAssembly mais simples e mais curto possível? Crie um arquivo nomeado como `texto.wat` e, dentro, preencha com o seguinte código:

```
(module)
```

Este módulo está totalmente vazio, mas ainda é um módulo válido. Agora podemos executar a ferramenta na linha de comando, como no exemplo a seguir:

```
# Se a configuração "-o" for omitida, o arquivo binário
# terá o mesmo nome do arquivo de texto.
#
# Neste código, por exemplo, se fosse omitida, iria criar
# um "texto.wasm" em vez de "modulo-simples.wasm".
wat2wasm texto.wat -o modulo-simples.wasm
```

Agora temos nosso arquivo `modulo-simples.wasm` criado no mesmo nível que o arquivo `texto.wat`. Mesmo sendo um módulo válido, ele não contém nenhum dado, função, tabela, nada além da definição módulo.

Formato binário para representação textual

A operação inversa da criação de um arquivo binário baseado em um arquivo de texto também é possível usando a ferramenta `wasm2wat`. Sua execução é baseada em receber um arquivo binário e imprimi-lo na linha de comando. Se você desejar criar o arquivo, é necessário usar a opção `-o` com o nome do arquivo desejado.

Para visualizar isso, podemos executar o comando `wasm2wat` usando o arquivo que acabamos de criar (`modulo-simples.wasm`), como mostra

o exemplo:

```
$ wasm2wat modulo-simples.wasm
(module)
```

A execução desse comando simplesmente retornaria o módulo que acabamos de escrever; porém, esse comando tem a tendência de criar um resultado diferente do `wat` original e isso se deve ao fato de a ferramenta reconhecer apenas os dados do formato binário.

Para um exemplo prático da diferença na reconversão para o formato de texto, considere que temos uma função que realiza subtração e divisão, como no código a seguir, salvo em um arquivo nomeado como `texto-subtracao-e-divisao.wat`:

```
(module
  (func $funcao (param i32) (param i32) (param i32) (result i32)
    (i32.sub (local.get 0)
      (i32.div_u (local.get 1) (local.get 2)
    ))
  )
  (export "subtracao-e-divisao" (func $funcao))
)
```

Nesse código, temos uma simples função que recebe três parâmetros e faz a subtração do primeiro com o valor da divisão dos dois valores restantes. Se fizermos a conversão desse código do formato de texto usando a ferramenta `wat2wasm`:

```
wat2wasm texto-subtracao-e-divisao.wat -o binario-subtracao-e-divisao.wasm
```

E, posteriormente, a conversão do arquivo binário gerado para o formato de texto usando a ferramenta `wasm2wat`:

```
$ wasm2wat binario-subtracao-e-divisao.wasm
(module
  (type (;0;) (func (param i32 i32 i32) (result i32)))
  (func (;0;) (type 0) (param i32 i32 i32) (result i32)
    local.get 0
    local.get 1
```

```
    local.get 2
    i32.div_u
    i32.sub)
  (export "funcao" (func 0)))
```

O resultado do formato de texto gerado do binário é completamente diferente do texto original. Isso se deve por dois motivos:

- O arquivo binário criado a partir de `texto-subtracao-e-divisao.wat` não mantém a fidelidade da estrutura do código de texto, como o uso dos parênteses, além da adição de tipos feita pelo compilador;
- A ferramenta `wasm2wat` reconhece apenas os dados binários e não tem a menor ideia ou referência de como o arquivo binário foi originalmente produzido, seja usando Rust, C++, C#, WAT ou qualquer outra forma de compilação.

Tendo em vista esses aspectos, não espere que a ferramenta ajude a depuração de arquivos binários sendo fiel ao código original.

Validação do formato binário

Se desejarmos validar arquivos binários, podemos usar uma das ferramentas binárias do WABT conhecida como `wasm-validate`. O uso da ferramenta na aplicação de linha de comando espera receber o caminho do arquivo binário a ser validado.

Veja um exemplo do uso da ferramenta com o módulo simples que escrevemos anteriormente:

```
wasm-validate modulo-simples.wasm
```

Não imprimirá nada na linha de comando, pois considera o arquivo binário como válido. Se houver qualquer erro no formato binário, os erros serão impressos na linha de comando.

> Nota: quando executado com o parâmetro `-v`, o comando tem como resultado a estrutura do arquivo binário.

Apesar de termos `wasm-validate` à nossa disposição, há também a possibilidade de validar e visualizar a estrutura de qualquer arquivo binário WebAssembly sem dependência de ferramentas, especialmente módulos pequenos como este que acabamos de criar.

10.2 Estrutura binária do WebAssembly

Se abrirmos o arquivo binário na maioria dos editores de texto, o conteúdo do binário estará escrito no formato do sistema hexadecimal:

```
0061 736d 0100 0000
```

Se o seu editor de texto exibir o arquivo binário em qualquer outro formato além do hexadecimal (por exemplo, número binário ou instruções *Assembly*), é possível exibi-lo em hexadecimal na linha de comando com o uso da ferramenta `hexdump`, disponível no OSX e na maioria das distribuições Linux:

```
hexdump -C modulo-simples.wasm
00000000  00 61 73 6d 01 00 00 00 |.asm....|
00000008
```

Se estiver usando Windows, existe a possibilidade de uso da ferramenta `Format-Hex`, disponível para versões do *powershell* acima do 5.0:

```
Format-Hex modulo-simples.wasm
```

O conteúdo do módulo de 8 bytes que acabamos de verificar é conhecido como o menor módulo WebAssembly possível e ele provê dois identificadores importantíssimos nos primeiros 64 bits:

```
0000000: 0061 736d              ; WASM_BINARY_MAGIC
0000004: 0100 0000              ; WASM_BINARY_VERSION
```

Os quatro primeiros bytes representam o valor do número mágico binário do WebAssembly: `\0asm`. O caractere `\0` é representado com o código `0`, ou seja, seria a dupla de zeros ("00") no valor "0061736d". Os três caracteres restantes são "a", "s" e "m", que são expressos pelos códigos ASCII "97", "115" e "109", respectivamente, ou "61", "73" e "6d" no formato do sistema hexadecimal, como vemos no exemplo.

Os próximos bytes representam a versão do WebAssembly usada em um formato de 32 bits. A versão do binário usada no exemplo é a 1.0 (`0x01`). Dependendo do destino de compilação que você possui, é comum ver essa versão mudar para outros valores; entretanto, os navegadores, por padrão, esperam a versão estável do WebAssembly, que, atualmente, é a 1.0.

Até então, não possuímos nada além da definição do módulo no `modulo-simples.wasm`. Que tal expandir o nosso módulo para ver os *bytes* que serão adicionados? Vamos começar escrevendo uma função nomeada `funcao` que retorna um inteiro de 32 bits com o valor constante de `85`.

```
(module
  (func (export "funcao") (result i32)
    i32.const 85
    return
  )
)
```

Salve o arquivo como `modulo-com-funcao.wat` e realize a compilação do código com o comando `wat2wasm modulo-com-funcao.wat`. Assim que você terminar de compilar o formato de texto, verá que o novo arquivo binário contém outras informações além do número mágico e a versão WebAssembly:

```
0061 736d 0100 0000 0105 0160 0001 7f03
0201 0007 0a01 0666 756e 6361 6f00 000a
```

```
0801 0600 41d5 000f 0b
```

Vamos explorar esse código binário byte por byte. No WebAssembly, os módulos são organizados em seções: seção de tipo, seção de função, seção de exportação, e outras. O primeiro byte de cada seção representa o identificador da seção. A primeira seção é definida após a declaração da versão do WebAssembly, onde os primeiros 2 bytes definem a configuração da seção.

No código binário anterior, os primeiros 2 bytes são "0105". A seção "0105", assim como todas as outras, é configurada pelo seu respectivo valor. Por exemplo: o valor "01" é responsável por identificar a seção como uma seção de tipo e o valor "05" define o tamanho, adicionando os próximos 5 bytes como parte da seção definida. Isso significa que a seção "0105" possui os respectivos 5 bytes seguintes: `0160 0001 7f` .

A seção de tipo essencialmente contém as assinaturas de funções. No código escrito em `modulo-com-funcao.wat` , temos uma função com zero parâmetros e um retorno do tipo inteiro de 32 bits.

A seguir, veja a leitura de cada um dos 5 bytes da seção de tipo:

```
01 ; Número de tipos usados (1)
60 ; Identificador da função (func)
00 ; Número de parâmetros (0)
01 ; Número de resultados (1)
7f ; Tipo do resultado (i32)
```

Após o processamento dos cinco bytes da seção de tipo, uma nova seção é iniciada. No nosso código binário, o próximo identificador é "03", que representa uma seção de função, seguida de seu respectivo tamanho, "02", que representa dois bytes.

A seção de função armazena índices da assinatura da função. Veja a leitura de cada um dos 2 bytes da seção de função definida:

```
01 ; Número de funções (1)
00 ; Indice da função (0)
```

Em seguida, a seção de exportação é definida com o uso do identificador "07". Esta seção define o nome e atributos de uma exportação com o índice com base na função definida anteriormente. Assim como as demais, ela é definida com seu respectivo tamanho, que neste caso é "0a", ou seja, 10 bytes (`0106 6675 6e63 616f 0000`).

```
; Número de exportações (1)
01

; Tamanho do nome exportado (6 bytes)
06

; Nome de função ("funcao")
6675 6e63 616f

; Tipo de exportação ("0" é o identificador do tipo função)
00

; Índice da função exportada (0)
00
```

A próxima seção é uma seção de código que é configurada com o identificador "0a" (10). A seção de código representa o código real da função. No nosso exemplo, esta seção recebe o tamanho de 8 bytes (`0106 0041 d500 0f0b`).

A seção de código tem uma característica "especial" que a difere das demais: as instruções do WAT que vimos no último capítulo são expressas em suas respectivas representações binárias. A instrução `i32.const` torna-se "41" e o valor "85" é codificado seguindo o padrão LEB128, assim com todos os inteiros no WebAssembly.

```
01   ; Número de funções (1)
06   ; Tamanho do corpo da função (6 bytes)
00   ; Número total de declarações locais (0)
41   ; Instrução "i32.const"
d500 ; Valor "85"
```

```
0f   ; Instrução "return"
0b   ; Fim do código da função
```

Terminamos a leitura passo a passo do nosso binário. Podemos concluir que o código binário é essencialmente dividido em um vetor de seções. A função que escrevemos posteriormente é distribuída em quatro seções: tipo, função, exportação e código. Se quiséssemos escrever um pseudocódigo que representasse as seções do nosso módulo, poderíamos escrever algo como:

```
Wasm magic number
version

section "type"
  func
  result type: i32

section "function"
  index of the function: 0

section "export"
  export name: "funcao"
  export kind: function
  index of the function: 0

section "code"
  i32.const
  i32 literal: 85
  return
```

No exemplo dado com apenas uma função que retorna um valor constante, fazer a leitura e interpretação do arquivo binário é bem simples, porém, quanto maior o binário, mais complexa fica a leitura etapa por etapa, como acabamos de fazer. Por esse motivo, o WABT provê outras ferramentas que facilitam a visualização do arquivo binário de forma automática.

10.3 Visualização de arquivos binários

A ferramenta `wasm-objdump` permite visualizar as seções do arquivo binário de forma automática. O uso do `wasm-objdump` como aplicação de linha de comando recebe o caminho do arquivo binário a ser visualizado. Para um teste rápido, podemos usar o arquivo `modulo-com-funcao.wasm` juntamente de todos os parâmetros que `wasm-objdump` nos provê, como `-dhxs` .

```
wasm-objdump -dhxs modulo-com-funcao.wasm
```

Os parâmetros `-dhxs` são uma mistura dos parâmetros de desmontagem (`-d`), contexto bruto (`-s`), configuração de detalhes (`-x`) e os cabeçalhos do arquivo binário (`-h`). Assim que o comando for executado, você terá todo o relatório das seções do módulo WebAssembly:

```
modulo-com-funcao.wasm:    file format wasm 0x1

Sections:

  Type start=0x0000000a end=0x0000000f (size=0x00000005) count: 1
 Function start=0x00000011 end=0x00000013 (size=0x00000002) count: 1
   Export start=0x00000015 end=0x0000001f (size=0x0000000a) count: 1
  Code start=0x00000021 end=0x00000029 (size=0x00000008) count: 1

Section Details:

Type[1]:
 - type[0] () -> i32
Function[1]:
 - func[0] sig=0 <funcao>
Export[1]:
 - func[0] <funcao> -> "funcao"
Code[1]:
 - func[0] size=6 <funcao>

Code Disassembly:
```

```
000023 func[0] <funcao>:
 000024: 41 d5 00                 | i32.const 85
 000027: 0f                       | return
 000028: 0b                       | end

Contents of section Type:
000000a: 0160 0001 7f                           .`...

Contents of section Function:
0000011: 0100                                   ..

Contents of section Export:
0000015: 0106 6675 6e63 616f 0000               ..funcao..

Contents of section Code:
0000021: 0106 0041 d500 0f0b                    ...A....
```

É bem legal gerar o relatório de todas as seções automaticamente, assim não precisamos contar byte por byte. O relatório de seções e objetos do `.wasm` é frequentemente usado para otimização, depuração e testes de binários.

O `wasm-objdump` é limitado à funcionalidade de visualização de estruturas binárias e, muitas vezes, não é suficiente para o processo de depuração, recorrendo ao uso de interpretadores. O pacote de ferramentas também provê um interpretador de binários conhecido como `wasm-interp`.

10.4 Execução de binários usando o interpretador

O `wasm-interp` faz a decodificação e execução de um arquivo binário usando um interpretador baseado em pilha. Você pode configurar esse comando com inúmeras opções, e cada opção reflete na

configuração do WebAssembly — por exemplo, habilitar o uso de *threads* (`--enable-threads`) ou até mesmo usar a memória baseada em 64 bits (`--enable-memory64`).

O interpretador permite também habilitar todas as funcionalidades do WebAssembly com o uso da opção `--enable-all` e ver todas as seções com suas respectivas configurações com o uso da opção `--verbose` .

```
wasm-interp --verbose modulo-com-funcao.wasm
```

O retorno do comando é similar ao `wasm-objdump` , mas é exibido de forma mais legível, sendo possível ver o tamanho das seções, nomes de funções e até os respectivos índices. Superinteressante!

```
BeginModule(version: 1)
  BeginTypeSection(5)
  OnTypeCount(1)
  OnFuncType(index: 0, params: [], results: [i32])
  EndTypeSection
  BeginFunctionSection(2)
  OnFunctionCount(1)
  OnFunction(index: 0, sig_index: 0)
  EndFunctionSection
  BeginExportSection(10)
  OnExportCount(1)
  OnExport(index: 0, kind: func, item_index: 0, name: "funcao")
  EndExportSection
  BeginCodeSection(8)
  OnFunctionBodyCount(1)
  BeginFunctionBody(0, size:6)
  OnLocalDeclCount(0)
  OnI32ConstExpr(85 (0x55))
  OnReturnExpr
  OnEndExpr
  EndFunctionBody(0)
  EndCodeSection
EndModule
   0| i32.const 85
```

```
  8| return
 12| return
```

Apesar de o exemplo acima ir um pouco além o `wasm-objdump` e entregar mais informações sobre as etapas da execução do binário, não há nenhuma execução de função exportada. Seria legal podermos ver a função `funcao` executada pelo interpretador e retornar o seu respectivo valor (85), e tal operação é possível!

A ferramenta `wasm-interp` pode executar todas as exportações definidas no arquivo binário em ordem com o uso da opção `--run-all-exports`, funcionalidade que é de extrema utilidade para testes. Se a opção `--run-all-exports` for usada em conjunto com a opção `--trace`, é possível acompanhar a execução de função por função na linha de comando.

```
$ wasm-interp --run-all-exports --trace modulo-com-funcao.wasm

>>> running export "funcao":
#0. 0: V:0  | i32.const 85
#0. 8: V:1  | return
funcao() => i32:85
```

Note que o comando retorna o nome da função executada juntamente do tipo retornado e seu respectivo valor. Existe uma limitação da opção `--run-all-exports`, pois ela não executará funções exportadas que possuem argumentos. Com base nessa limitação, o interpretador do WebAssembly disponibiliza outra possibilidade de execução de funções exportadas baseada na definição de funções e argumentos para serem executados (curiosidade: foi o autor deste livro que implementou essa funcionalidade no interpretador).

Para vermos um exemplo, vamos considerar um módulo WebAssembly que possui uma função exportada nomeada como `fatorial`, que executa uma função interna nomeada como `$calcfat` e recebe um valor `i32` para calcular o número fatorial:

```
(module
  (func (export "fatorial") (param i32) (result i32)
```

```
    local.get 0
    call $calcfat)

  (func $calcfat (param i32) (result i32)
    local.get 0
    i32.const 0
    i32.gt_s
    if (result i32)
      local.get 0
      local.get 0
      i32.const 1
      i32.sub
      call $calcfat
      i32.mul
      return
    else
      i32.const 1
      return
    end)
)
```

Salve esse código como `fatorial.wat` e realize a compilação com o `wat2wasm` usando o seguinte comando:

```
wat2wasm fatorial.wat
```

Se executarmos o interpretador usando o arquivo binário e configurando a função exportada `fatorial` com seu respectivo parâmetro, podemos obter o valor pelo próprio interpretador:

```
wasm-interp fatorial.wasm --run-export=fatorial --argument=i32:10
```

Após a execução do comando, vamos ter o resultado da função na linha de comando:

```
func_fac(i32:10) => i32:3628800
```

Perfeito! A função está retornando o resultado esperado (`10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 3628800`). Desta forma, podemos realizar

testes rápidos em uma ou múltiplas funções do arquivo binário WebAssembly.

Tanto o `wasm-objdump` como o `wasm-interp` podem fornecer visualizações (em diferentes níveis) de um arquivo binário; no entanto, a visualização gerada está longe de ser um código legível para quem não conhece WebAssembly.

10.5 Descompilação de binários

A ferramenta binária conhecida como `wasm-decompile` é um descompilador de arquivos binários. Essa ferramenta produz um código similar à linguagem de programação C baseada no arquivo binário recebido.

O `wasm-decompile` é extremamente útil para visualização de binários para programadores que procuram uma leitura simplificada da lógica do arquivo binário ou não possuem conhecimento algum sobre o WebAssembly.

A fim de exemplificarmos o poder dessa ferramenta, vamos escrever outro arquivo de texto para compilar para o formato binário. Nomearemos como `modulo-aritmetico.wat` e colocaremos 4 funções que serão exportadas: `soma`, `subtracao`, `divisao` e `multiplicacao`. Veja o código de implementação do módulo:

```
(module
  (func (export "soma") (param $lhs i32) (param $rhs i32) (result i32)
    local.get $lhs
    local.get $rhs
    i32.add
    return
  )
  (func (export "subtracao") (param $lhs i32) (param $rhs i32) (result
i32)
    local.get $lhs
```

```
    local.get $rhs
    i32.sub
    return
  )
  (func (export "divisao") (param $lhs i32) (param $rhs i32) (result i32)
    local.get $lhs
    local.get $rhs
    i32.div_u
    return
  )
  (func (export "multiplicacao") (param $lhs i32) (param $rhs i32) (result
i32)
    local.get $lhs
    local.get $rhs
    i32.mul
    return
  )
)
```

Assim que o arquivo de texto for criado, executaremos diretamente sua compilação para o arquivo binário usando o seguinte comando:

```
wat2wasm modulo-aritmetico.wat
```

Agora que temos nosso arquivo binário em mãos, vamos descompilar usando o comando `wasm-decompile`. Se o parâmetro de configuração do nome do arquivo de saída não for usado, a ferramenta `wasm-decompile` usará o mesmo nome do arquivo binário junto à extensão de arquivo `.c`.

Neste exemplo de comando, note que estamos nomeando o arquivo de saída como `modulo-aritmetico-descompilado.c`:

```
wasm-decompile modulo-aritmetico.wasm -o modulo-aritmetico-descompilado.c
```

Um novo arquivo foi gerado justamente com o nome que escolhemos; se você o abrir, verá que parece muito com a linguagem de programação C, mas não é exatamente C:

```
export function soma(a:int, b:int):int {
  return a + b
}

export function subtracao(a:int, b:int):int {
  return a - b
}

export function divisao(a:int, b:int):int {
  return a / b
}

export function multiplicacao(a:int, b:int):int {
  return a * b
}
```

O comando `wasm-decompile` auxilia muito a depuração de arquivos binários do WebAssembly já existentes, permitindo a leitura de forma rápida e prática dos conteúdos (como funções, tipos, tabelas e outros) que foram especificados no arquivo binário.

Apesar de ser um código lógico legível para qualquer pessoa programadora que não conhece WebAssembly, ainda assim é um pseudocódigo e não pode ser executado.

Se quiséssemos subir um degrau a mais no nível de descompilação do arquivo binário para, de fato, um executável válido, poderíamos usar outra ferramenta do pacote binário, conhecida como `wasm2c`.

10.6 wasm2c: conversão de arquivos binários para C

A ferramenta `wasm2c` permite criar um arquivo de código válido da linguagem de programação C, junto ao seu respectivo arquivo de cabeçalho, que possui a extensão do arquivo em `.h`. Seu conteúdo

possui a definição das estruturas, assinatura das funções e procedimentos do arquivo C criado.

É um degrau acima do `wasm-decompile`, pois providencia código lógico válido para a compilação. Apesar de o código ser executável, é certo afirmar que o utilitário `wasm2c` dificultará a leitura rápida de um módulo, pois adiciona demasiado código em ordem do funcionamento correto do módulo com as especificações do WebAssembly aplicados na linguagem C.

Porém, mesmo afetando o tamanho do código gerado e, consequentemente, a leitura do módulo, a depuração ocorre em um nível que o `wasm-decompile` não consegue fornecer, permitindo depurar a representação do módulo em tempo de execução e, em certos casos, no tempo de compilação.

A ferramenta `wasm2c` pode ser usada assim como as demais ferramentas binárias que vimos até agora. A maioria das ferramentas do WABT disponibiliza o uso de configurações para ativar funcionalidades do WebAssembly, como *threads*, memória em 64 bits e outras.

No caso da ferramenta `wasm2c`, o uso dessas opções tem um peso enorme, podendo mudar a complexidade do código gerado. Então, é claro que qualquer dessas opções só deve ser usada se de fato seu módulo foi criado visando recursos específicos do WebAssembly.

> Certas opções, como habilitar funcionalidade e suporte de *threads*, *tail call* ou *annotation*, não funcionarão no `wasm2c` por falta de suporte. É possível ver a lista completa de funcionalidades suportadas em: https://github.com/webAssembly/wabt#supported-proposals.

O `wasm2c`, quando executado na linha de comando, não vai criar um arquivo por padrão, e diferentemente dos comandos que vimos até então, o parâmetro `-o` já não é mais opcional se desejarmos criar

um arquivo após a execução do comando. Veremos a seguir um exemplo de compilação para código C junto com o seu respectivo arquivo de cabeçalho. Execute o seguinte comando:

```
wasm2c modulo-aritmetico.wasm -o modulo-aritmetico.c
```

Após a execução desse comando, dois arquivos serão criados: `modulo-aritmetico.c` e `modulo-aritmetico.h`. Você vai perceber que o conteúdo do `modulo-aritmetico` é bem maior do que o arquivo `modulo-aritmetico-descompilado.c`. É importante destacar que o resultado do compilador muda de acordo com a sua versão do WABT.

A seguir, temos a visualização da função `soma` em diferentes partes do arquivo `modulo-aritmetico.c`:

```
static u32 w2c_soma(Z_moduloZ2Daritmetico_instance_t*, u32, u32);

// ...

static u32 w2c_soma(Z_moduloZ2Daritmetico_instance_t* instance, u32
w2c_p0, u32 w2c_p1) {
  FUNC_PROLOGUE;
  u32 w2c_i0, w2c_i1;
  w2c_i0 = w2c_p0;
  w2c_i1 = w2c_p1;
  w2c_i0 += w2c_i1;
  goto w2c_Bfunc;
  w2c_Bfunc:;
  FUNC_EPILOGUE;
  return w2c_i0;
}

// ...

/* export: 'soma' */
u32 Z_moduloZ2DaritmeticoZ_soma(Z_moduloZ2Daritmetico_instance_t*
instance, u32 w2c_p0, u32 w2c_p1) {
  return w2c_soma(instance, w2c_p0, w2c_p1);
}
```

Note que a função `soma` usa diversos recursos que só existem no contexto criado pela ferramenta do `wasm2c`. O código pode ser compilado pelo compilador da linguagem C e posteriormente executado, se desejado.

Até então, todos os pacotes e recursos que vimos pertencem ao WABT e permitem trabalhar com código no formato de texto e binário. Entretanto, ainda existe mais um pacote de ferramentas binárias oficial do WebAssembly: o *Binaryen*.

10.7 Binaryen

Nesta seção, conversaremos sobre o Binaryen e também sobre otimizações que podemos fazer no compilador do Rust usando Cargo. O Binaryen é um pacote de ferramentas conhecido por focar majoritariamente na geração e otimização de arquivos binários de alta performance.

O uso e propósito do Binaryen é diferente do WABT, pois o WABT foca exclusivamente na especificação e não fornece uma plataforma de otimização ou um destino de compilação de alto nível.

> Curiosidade: o nome *Binaryen* vem da junção da palavra *binary* (binário, em inglês) com *Emscripten*. O criador do Emscripten, Alon Zakai, é atualmente um dos principais contribuidores e mantenedores do Binaryen.

As ferramentas disponíveis pelo Binaryen possuem operações para reduzir (quando possível) o tamanho do arquivo binário, como também para remover complexidade lógica desnecessária adicionada por compiladores. São um conjunto abrangente de ferramentas com infraestrutura de suporte para uso como *back-end* de compiladores que visam o WebAssembly como formato de saída. O Binaryen

possui uma interface em C e implementa sua própria representação intermediária interna (termo abreviado como *IR* e conhecido como *Intermediate Representation* em inglês) da lógica do programa, podendo realizar várias otimizações pelo Binaryen IR. Uma representação intermediária interna é a linguagem de uma máquina abstrata projetada para auxiliar na análise de programas de computador. O termo vem de seu uso em compiladores, onde o código-fonte de um programa é traduzido para uma forma mais adequada, que viabiliza transformações de melhoria de código antes de ser usado para gerar código de objeto ou máquina.

A Binaryen IR é usada como parte do compilador *asm2wasm*, que pode converter arquivos *asm.js* em arquivos WebAssembly. Outro caso de uso que vale destaque é para auxiliar a infraestrutura da LLVM no compilador do Rust para gerar WebAssembly. Compiladores, essencialmente, fazem uso de técnicas avançadas de otimização e, por isso, é bem comum o uso do Binaryen e de seu respectivo pacote de ferramentas em inúmeros compiladores. Dito isso, é certo concluir que, em termos de qualidade, os binários produzidos com o Binaryen são de "primeira mão".

A seguir, veja algumas das funcionalidades do Binaryen como aplicação de linha de comando:

- `wasm-opt` : otimiza qualquer arquivo binário com o processamento do Binaryen IR;
- `wasm-as` : compila o formato de texto passando pelo Binaryen IR em um arquivo de formato binário. Funciona de forma similar ao `wat2wasm` do WABT; sua única diferença é que se prende menos à especificação do WebAssembly e é exigente quanto ao arquivo de texto. É normal disparar algum tipo de erro que não existiria usando o WABT;
- `wasm-dis` : desmonta o WebAssembly em formato binário para o formato de texto, também processado pelo Binaryen IR. Realiza a mesma funcionalidade do `wasm2wat` do WABT;
- `wasm-shell` : um *shell* que pode carregar e interpretar o código WebAssembly (um interpretador de comandos é conhecido

como *shell* ou modo texto);
- `binaryen.js` : uma biblioteca JavaScript independente que expõe métodos do Binaryen para criar e otimizar módulos WebAssembly. A instalação do `binaryen.js` pode ser feita via NPM ou diretamente da origem do pacote. A biblioteca permite sua execução no NodeJS e em diversos navegadores.

> Você pode ver a lista completa das ferramentas no repositório oficial do Binaryen no GitHub, disponível em: https://github.com/WebAssembly/binaryen.

Todas as ferramentas do pacote possuem objetivos diferentes, mas compartilham do processamento com o Binaryen IR. Antes de começarmos a usar as ferramentas, vamos conversar primeiro sobre como o Binaryen IR realmente funciona e quais são os tipos de otimização aplicadas.

O IR interno do Binaryen foi projetado para ser flexível e rápido para otimização. Sua estrutura é o mais próximo possível do WebAssembly, portanto, é simples e rápido convertê-lo para o formato de texto ou para o formato binário.

O Binaryen contém muitas etapas de otimização para tornar o WebAssembly menor e mais rápido. As etapas de otimização contém:

- Eliminação de funções, parâmetros e valores inutilizados;
- Transformação de chamadas indiretas (`call_indirect`) em uma chamada normal (`call` , quando o índice da tabela é constante);
- Remoção de duplicatas no código com a aplicação de "dobragem de código" (*Code Folding*, em inglês);
- Junção de um bloco a um bloco externo sempre que possível, reduzindo o número de blocos;
- Junção de variáveis locais que possuem o mesmo valor;
- Aplicação de *Inlining*, um processo de substituição de uma sub-rotina ou uma chamada de função pelo corpo da sub-rotina ou

função que está sendo chamada;
- Eliminação de subexpressões comuns (em inglês, *"Common subexpression elimination"*, abreviado como CSE). Essa otimização procura por instâncias de expressões idênticas (ou seja, todas elas avaliam o mesmo valor) e analisa se vale a pena substituí-las por uma única variável contendo o valor calculado;
- Minificação de importações e exportações, por exemplo, renomeia para "r" ou "v";
- Pré-computação de valores: calcula expressões constantes em tempo de compilação usando o interpretador integrado, que garante a capacidade de lidar com qualquer expressão constante;
- Simplificação dos globais e locais em diferentes formas. Por exemplo, se um valor global mutável nunca é modificado, o valor global é alterado para constante (imutável).

Existem muitas outras otimizações além das listadas aqui. Se você desejar ver a lista de otimizações, basta acessar o link do repositório oficial do Binaryen: https://github.com/WebAssembly/binaryen.

O Binaryen é instalado separadamente do WABT. Para instalar as ferramentas do Binaryen como aplicação de linha de comando, siga a respectiva instrução com base no seu sistema operacional:

- OS X usando *Homebrew* como gerenciador de pacotes: `brew install binaryen` ;
- Debian: `apt-get install binaryen` ;
- Ubuntu: `apt-get install binaryen` ;
- Arch Linux: `pacman -S binaryen` ;
- Kali Linux: `apt-get install binaryen` ;
- Windows (*WSL2*): `apt-get update && apt-get install binaryen` .

Após a instalação do Binaryen, algumas novas ferramentas de manipulação do WebAssembly estarão acessíveis para uso. Na próxima seção, vamos conversar sobre a otimização de arquivos binários.

10.8 Otimização de arquivos binários

Qualquer módulo WebAssembly criado com o compilador do Rust ou usando as ferramentas do WABT é suscetível a otimizações de performance pelo Binaryen. Todavia, se as otimizações serão efetivas e o quão efetivas serão depende de como o binário foi configurado e criado.

Que tal vermos sobre como as otimizações funcionam de forma prática? Vamos voltar ao editor de imagens que escrevemos no quarto capítulo e inspecionar o tamanho do arquivo binário criado dentro do caminho de pastas `target/wasm32-unknown-unknown/release` :

```
$ du -h ./target/wasm32-unknown-unknown/release/editor.wasm
1.7M  ./target/wasm32-unknown-unknown/release/editor.wasm
```

O tamanho do arquivo binário WebAssembly é 1.7 Megabytes. Bem grandinho, né? Porém, não definimos nenhuma otimização na compilação usando Cargo para o perfil de `release` , ou seja, se executarmos um singelo `cargo build` , o arquivo gerado terá quase o mesmo tamanho (1.8 Megabytes):

```
# Compilação sem "--release" irá criar o binário na pasta "debug"
$ cargo build

$ du -h ./target/wasm32-unknown-unknown/debug/editor.wasm
1.8M  ./target/wasm32-unknown-unknown/debug/editor.wasm
```

O `cargo` possui 4 perfis de compilação: *dev*, *release*, *bench* e *test*. Os quatro perfis possuem configurações diferentes e têm propósitos diferentes:

- O perfil *dev* é usado para desenvolvimento normal e depuração;
- O perfil *release* (*lançamento*, em inglês) destina-se a artefatos otimizados usados para produção;
- O perfil *test* é o perfil padrão usado pelo `cargo test` ;
- O perfil *bench* é o perfil padrão usado pela bancada de carga.

No caso do editor, o binário que produzimos é considerado imenso quando comparado com a quantidade lógica existente do editor de imagens. Isso significa que a configuração de saída do perfil *release* não foi otimizada. A fim de reduzir o tamanho do arquivo binário do editor de imagens, podemos adicionar otimizações específicas; para isso, precisamos atualizar o `Cargo.toml` do projeto com o seguinte código:

```
[profile.release]
lto = true
codegen-units = 1
opt-level = 's'
```

Nesse código, estamos especificando e habilitando algumas etapas de otimização.

Você pode ver a configuração de valores padrões para cada perfil do Cargo em sua documentação oficial: https://doc.rust-lang.org/cargo/reference/profiles.html#default-profiles.

A primeira configuração habilitada é o `lto` (do inglês, *Link Time Optimization*), uma forma de otimização interprocedural que é realizada no momento da vinculação do código do aplicativo, permitindo que todas as diferentes unidades de compilação que compõem um único executável sejam otimizadas como um único módulo. Habilitar a configuração `lto` pode afetar consideravelmente o tempo de compilação, pois é uma operação computacional "cara".

A segunda configuração é a especificação do número das unidades de geração de código para apenas uma. O compilador do Rust (`rustc`) pode dividir automaticamente sua representação intermediária (LLVM IR) em várias unidades de compilação, chamadas de "unidades codegen".

O grau em que é eficaz depende muito de como as dependências internas de um *crate* são organizadas e da capacidade do compilador de entendê-las. Isso pode resultar em uma recompilação

parcial mais rápida, em detrimento da otimização, pois as oportunidades de *inlining* são perdidas.

A terceira e última otimização é a configuração `opt-level`. Ela controla o nível da opção `-C` no compilador do Rust, que controla o nível de otimização. Isso vai refletir em diversos aspectos: níveis de otimização mais altos podem produzir código de tempo de execução mais rápido às custas de tempos de compilação mais longos. Níveis mais altos também podem alterar e reorganizar o código compilado, o que pode dificultar o uso com um depurador.

Veja os níveis de otimização da configuração `opt-level`:

- `opt-level = '0'`: nenhuma otimização. Este nível é o padrão do perfil *dev* e *test*;
- `opt-level = '1'`: otimizações de nível básico;
- `opt-level = '2'`: otimizações de nível básico com otimizações especiais;
- `opt-level = '3'`: todas as otimizações possíveis. Este nível é o padrão dos perfis *release* e *bench*;
- `opt-level = 's'`: otimizações para reduzir o tamanho do arquivo binário;
- `opt-level = 'z'`: otimizações para reduzir o tamanho do arquivo binário, porém desabilita a funcionalidade de transformação em que a mesma operação é executada ao mesmo tempo em vários elementos vetoriais (essa otimização é conhecida como vetorização de *loops*, do inglês, *loop vectorization*).

Salve o `Cargo.toml` e execute o comando `cargo build --release` na raiz do editor de imagens. Agora, se inspecionarmos o tamanho do arquivo binário criado dentro do caminho de pastas `target/wasm32-unknown-unknown/release`:

```
$ du -h target/wasm32-unknown-unknown/release/editor.wasm
48K  ./target/wasm32-unknown-unknown/release/editor.wasm
```

Grande diferença! O tamanho do arquivo binário foi de 1.7 Megabytes a 48 Kilobytes; baseado em uma média de tamanho de

arquivos baixados pelo navegador, atualmente podemos assumir que não é considerado um arquivo pesado (essa afirmativa depende de aplicação para aplicação).

Podemos reduzir ainda mais o tamanho do binário com as otimizações do Binaryen. Para isso, precisamos da ferramenta `wasm-opt`.

Enxugando arquivos binários com Binaryen

A ferramenta `wasm-opt` permite aplicar otimizações em um arquivo binário. A diferença do `wasm-opt` para as outras ferramentas do Binaryen é que ele permite especificar quais otimizações serão aplicadas em um arquivo binário.

Existem várias opções que você pode definir para ajustar os níveis de otimização e redução, como: ignorar armadilhas improváveis, heurísticas em linha, matemática rápida e assim por diante. Para saber como configurá-los e outros detalhes, consulte o comando: `wasm-opt --help`.

Por ora, vamos com as otimizações padrões do Binaryen. Para isso, precisamos usar o parâmetro `-Oz` (maiúsculo) juntamente do caminho binário que estamos otimizando e o arquivo de saída (posterior ao parâmetro `-o` em minúsculo). O parâmetro `-Oz` executa as otimizações padrões, focando no tamanho do código:

```
$ wasm-opt -Oz ./target/wasm32-unknown-unknown/release/editor.wasm -o
editor-binaryen.wasm

$ du editor-binaryen.wasm
40K editor-binaryen.wasm
```

Apenas a execução das otimizações padrões reduziu aproximadamente 8 Kilobytes do arquivo binário. É provável que possamos reduzir até um pouco mais elevando o nível de otimização com o uso dos parâmetros `-01`, `-02`, `-03` ou `-04`, que adicionam mais etapas e podem tornar o processamento mais lento. Neste

caso, o uso das otimizações padrões do Binaryen afetaram o tamanho do arquivo em *16.67%*.

No entanto, as reduções do arquivo variam com o tamanho do módulo: quanto maior o módulo, mais efetivo o Binaryen se torna. No caso de grandes módulos WebAssembly, as otimizações podem cortar mais da metade do tamanho do arquivo binário, dependendo das configurações utilizadas, código e dependências do módulo escrito.

Outra informação que vale o destacar é que compiladores que destinam binários em WebAssembly possuem implementações distintas e isso reflete no tamanho do binário produzido por diferentes linguagens. O compilador do Rust, quando configurado corretamente, faz um excelente trabalho em manter a performance próxima das otimizações do Binaryen, mas compiladores de outras linguagens podem gerar um binário excessivamente longo.

Agora que vimos sobre como a otimização de binários funciona usando a ferramenta `wasm-opt`, vamos ver outra ferramenta que o Binaryen provê, mas que não é focada em otimização e, sim, na depuração de módulos WebAssembly em tempo de execução.

10.9 wasm2js: conversão de arquivos binários para JavaScript

O `wasm2js` é uma ferramenta que nos permite criar módulos JavaScript ES2015 com base em código binário do WebAssembly. Frequentemente usada para depurar arquivos binários em tempo de execução, ela permite aos desenvolvedores a possibilidade de escrever uma suíte de testes usando JavaScript tanto nos navegadores como em servidores (usando NodeJS).

O `wasm2js` funciona de forma muito parecida com o `wasm2c` do WABT e provê ainda mais opções de configuração de saída. Para ver todas as opções disponíveis, use o comando `wasm2js --help`. Tal qual as outras ferramentas vistas anteriormente, é preciso definir o nome e a extensão do arquivo de saída, pois sua saída por padrão é impressão do resultado na linha de comando.

O comando a seguir realiza a conversão do arquivo binário `modulo-aritmetico.wasm` em um módulo ES2015 JavaScript:

```
wasm2js modulo-aritmetico.wasm -o modulo-aritmetico.mjs
```

Após a execução desse comando, o arquivo `modulo-aritmetico.mjs` será criado e seu conteúdo deverá ser similar a este próximo código, podendo ter pequenas variações baseadas na versão instalada do Binaryen:

```
function asmFunc(imports) {
 var Math_imul = Math.imul;
 var Math_fround = Math.fround;
 var Math_abs = Math.abs;
 var Math_clz32 = Math.clz32;
 var Math_min = Math.min;
 var Math_max = Math.max;
 var Math_floor = Math.floor;
 var Math_ceil = Math.ceil;
 var Math_trunc = Math.trunc;
 var Math_sqrt = Math.sqrt;
 function $0($0_1, $1_1) {
  $0_1 = $0_1 | 0;
  $1_1 = $1_1 | 0;
  return $0_1 + $1_1 | 0 | 0;
 }

 function $1($0_1, $1_1) {
  $0_1 = $0_1 | 0;
  $1_1 = $1_1 | 0;
  return $0_1 - $1_1 | 0 | 0;
 }
```

```
 function $2($0_1, $1_1) {
  $0_1 = $0_1 | 0;
  $1_1 = $1_1 | 0;
  return ($0_1 >>> 0) / ($1_1 >>> 0) | 0 | 0;
 }

 function $3($0_1, $1_1) {
  $0_1 = $0_1 | 0;
  $1_1 = $1_1 | 0;
  return Math_imul($0_1, $1_1) | 0;
 }

 return {
  "soma": $0,
  "subtracao": $1,
  "divisao": $2,
  "multiplicacao": $3
 };
}

var retasmFunc = asmFunc({
});
export var soma = retasmFunc.soma;
export var subtracao = retasmFunc.subtracao;
export var divisao = retasmFunc.divisao;
export var multiplicacao = retasmFunc.multiplicacao;
```

Perceba que o arquivo criado é bem menor que a versão em C, porém ainda maior que o resultado do `wasm-decompile`.

A validação do módulo ES6 que acabamos de criar pode ser realizada no navegador ou usando NodeJS. Vamos usar a opção de teste do navegador, porém, se você desejar testar posteriormente com NodeJS, o exemplo está disponível no repositório do GitHub, em https://github.com/raphamorim/livro-webassembly-exemplos.

Para testar com o módulo ES6 criado, basta criar um arquivo `index.html` com o código a seguir e inicializar um servidor local,

como fizemos anteriormente no decorrer do livro:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Exemplo</title>
</head>
<body>
  <script type="module" src="modulo-aritmetico.mjs"></script>
  <script type="module">
  import { soma, divisao, subtracao, multiplicacao } from "./modulo-
aritmetico.mjs";
  console.log('A soma de 8 e 5 é ', soma(8, 5));
  console.log('A divisão de 5 e 5 é ', divisao(5, 5));
  console.log('A subtração de 85 e 85 é ', subtracao(85, 85));
  console.log('A multiplicação de 5 e 8 é ', multiplicacao(5, 8));
  </script>
</body>
</html>
```

Veja o resultado do módulo ES6 no navegador:

Figura 10.1: Resultado do módulo ES6 no navegador.

A possibilidade de ter um módulo ES6 criado a partir de um arquivo binário WebAssembly permite executar a representação da lógica do módulo em navegadores ou no NodeJS e se torna útil para escrever testes em plataformas que usam JavaScript como linguagem de programação.

10.10 wasm-tools

Além do Binaryen e do WABT, existe outro pacote de ferramentas binárias conhecido como `wasm-tools`, que, apesar de não ser um projeto oficial do grupo do WebAssembly, é mantido pelo grupo Bytecode Alliance, uma organização sem fins lucrativos dedicada à criação de novas fundações de software seguras, baseadas em padrões como o WebAssembly e o WASI (que vamos ver no próximo capítulo).

Diversas ferramentas providas pelo Binaryen e pelo WABT existem no `wasm-tools`, porém com características diferentes. Por exemplo, o WABT tem atualmente o melhor conjunto de ferramentas dentro da especificação oficial, assim como o Binaryen tem a melhor otimização entre os demais pacotes.

De certa forma, o `wasm-tools` é uma versão mais moderna dos demais pacotes de ferramentas binárias, além de ser escrito em Rust (WABT e Binaryen são escritos em C em C++), permitindo usar diferentes funcionalidades em um pacote do Rust. O `wasm-tools` pode ser instalado e usado diretamente na linha de comando com o uso do `cargo`. Mais informações no repositório oficial: https://github.com/bytecodealliance/wasm-tools.

Não vamos exemplificar o uso de todas as ferramentas do `wasm-tools` (você pode ver a lista de todas as ferramentas disponíveis também no link do repositório oficial), mas para ter uma ideia de quão similar o `wasm-tools` é em termos de funcionalidades, vamos fazer um rápido experimento. Instale o `wasm-tools` usando `cargo` com o seguinte comando:

```
cargo install wasm-tools
```

Após a instalação, você pode usar qualquer ferramenta do pacote seguindo o padrão `wasm-tools <nome-da-ferramenta>`, diferentemente do WABT e Binaryen, nos quais acessamos a ferramenta diretamente. Se desejássemos, por exemplo, executar a operação similar ao `wasm2wat` do WABT, podemos usar o comando `wasm-tools print`. Veja um exemplo de uso com o arquivo `binario-subtracao-e-divisao.wasm`, que geramos anteriormente:

```
$ wasm-tools print binario-subtracao-e-divisao.wasm
(module
  (type (;0;) (func (param i32 i32 i32) (result i32)))
  (func (;0;) (type 0) (param i32 i32 i32) (result i32)
    local.get 0
    local.get 1
    local.get 2
```

```
    i32.div_u
    i32.sub
  )
  (export "funcao" (func 0))
)%
```

O resultado do comando é exatamente idêntico ao resultado que obtemos anteriormente usando `wasm2wat`, porém a saída do comando tende a ter pequenas (ou até significantes) mudanças com base no tamanho do código do binário usado, já que ambas as ferramentas seguem abordagens diferentes — o IR do `wasm-tools` tem regras bem diferentes do Binaryen IR.

O maior valor do `wasm-tools` em comparação com as outras ferramentas é a simplicidade de uso no universo do Rust. O WABT e o Binaryen podem também ser usados pelo Rust, porém é um pouco mais trabalhoso, já que as funções precisam ser mapeadas na vinculação de C para Rust.

Com isso, encerramos nosso capítulo de estruturas binárias, WABT, Binaryen e, adicionalmente, `wasm-tools`. No próximo capítulo, estudaremos como funcionam as interfaces de sistema do WebAssembly, também conhecida como WASI, que permite executar módulos no servidor *back-end* realizando operações como leitura e escrita de arquivo, *IO*, requisições HTTP, entre outras.

Capítulo 11
WASI - WebAssembly System Interface

Finalmente, chegamos ao último capítulo do livro e, antes de começarmos, queria parabenizar a você, querido(a) leitor(a). WebAssembly já não é mais um estranho e agora você tem a noção básica e intermediária para construir qualquer programa com WASM, independentemente de linguagem ou ferramenta utilizada. Este capítulo em especial traz uma nova perspectiva de uso do WebAssembly, em comparação com tudo que vimos no decorrer do livro. Estudaremos sobre o WASI, que abre um outro mundo de possibilidades de uso com o WebAssembly.

11.1 O que é o WASI?

O acrônimo WASI vem do inglês para *WebAssembly System Interface*, que, traduzido para o português, seria algo como *interface de sistema do WebAssembly*. O WASI é um grupo de APIs que está sendo projetado por um subgrupo da comunidade WebAssembly na W3C, especialmente pelos integrantes do projeto Wasmtime, para ser proposto como padrões que operam de forma independente de qualquer plataforma e não são orientados ao universo Web.

O WebAssembly foi originalmente desenhado e projetado para rodar bem e, também, resolver certos problemas existentes da Web, mas não está limitado somente à ela. O WebAssembly interage com o mundo externo exclusivamente por meio de APIs e sua principal linguagem é independente de seu ambiente circundante (como vimos no capítulo 9, *Destrinchando o formato de texto*, o WAT). Na Web, o WASM usa naturalmente as APIs da Web existentes fornecidas pelos navegadores. No entanto, fora dos navegadores, atualmente não há um conjunto padrão de APIs para o uso de

programas WebAssembly e essa ausência dificulta a criação de programas para fora da Web.

O WASI é uma iniciativa para preencher essa lacuna, com um conjunto de APIs padronizadas que podem ser implementadas em várias plataformas, não sendo dependentes de nenhuma funcionalidade de um navegador (embora ainda possam ser executadas em navegadores). Todas as propostas de padrões do WASI têm algo em comum: permitir que o WebAssembly seja executado fora da Web. É possível usar WASM no *back-end* de formas distintas, como o instanciamento de módulos pela API do JavaScript no NodeJS — uma vez possuindo uma camada abaixo do WebAssembly para que seja executado no *back-end*. No cenário descrito, é necessário prover uma plataforma base que é responsável pelas APIs de sistema, ou seja, se você quiser ler um arquivo e processar com WebAssembly no NodeJS, seria obrigatório fazer o processo do *I/O* (do inglês, *in* e *out*) com as APIs do NodeJS e, posteriormente, trafegar os dados lidos para o WASM.

Já com o uso do WASI, o NodeJS se torna irrelevante nesse cenário, uma vez que é possível usar WebAssembly como a camada primária para operações I/O e os dados não precisam ser trafegados, pois, assim que o arquivo fosse lido, os dados já estariam dentro do WebAssembly. Como dito anteriormente, o WASI é uma família de APIs e cada uma provê diferentes funcionalidades. A funcionalidade de arquivos pertence à proposta de I/O (atualmente em estágio 2), mas, anteriormente, fazia parte do *WASI Core*.

O WASI Core surgiu em 2018 e era o principal objetivo do WASI em termos de padronização de APIs. Era um módulo que abrangia APIs de vários recursos do sistema operacional, incluindo arquivos e sistemas de arquivos, *Berkeley sockets* (também conhecidos como soquetes da Berkeley), relógios e números aleatórios.

Berkeley sockets é uma interface de programação de aplicativos para *sockets* de internet e *sockets* de domínio Unix, usados para comunicação entre processos (IPC) da Berkeley Software Distribution (BSD). É comumente implementado como uma biblioteca de módulos vinculáveis. Originou-se com o sistema operacional 4.2BSD Unix, lançado em 1983.

Eventualmente, as APIs do WASI Core foram dissolvidas e o termo já não é tão comum na comunidade WebAssembly. Anteriormente, as propostas de APIs eram agrupadas com base em sua funcionalidade, mas agora cada API tem sua própria proposta. A seguir, veja algumas das propostas do WASI com uma breve explicação:

- API de I/O: lida com *streaming* de arquivos, *sockets* e *pipes*;
- API de sistema de arquivos: lida com o sistemas de arquivos. Possui funções para abrir, ler e gravar arquivos e também para trabalhar com diretórios;
- API de relógio: lida com a leitura da hora atual e medição do tempo decorrido;
- API de aleatoriedade: lida com a obtenção de dados pseudoaleatórios;
- API de *Polling*: lida com a espera de eventos de I/O. *Polling* é o ato de constantemente monitorar ou esperar por atualizações de eventos. Os eventos podem vir de canais de dados, descritores de arquivos e outros;
- API de aprendizado de máquina (*Machine Learning*, em inglês): lida com inferência de aprendizado de máquina. Seu nome deriva do fato de que os modelos de aprendizado de máquina também são conhecidos como redes neurais (em inglês, *neural networks*, ou NN) ;
- API de *sockets*: lida com *sockets* TCP, UDP e pesquisa de nome de domínio. Essa proposta adiciona a interface básica de um *socket* BSD;
- API de *threads*: esta proposta é um passo à frente da API de *threads* do WebAssembly. Como vimos no capítulo 7, a proposta

do WebAssembly fornece as primitivas necessárias para memória compartilhada, operações atômicas, espera e notificação. A proposta do WASI disponibiliza apenas um mecanismo para gerar *threads*;
- API de *SQL*: adiciona uma interface que permite que programas escritos em WebAssembly possam interagir com bancos de dados SQL de forma genérica e segura.

Existem outras propostas do WASI não listadas aqui. Se você desejar ver a lista completa das propostas (em inglês), acesse o repositório oficial no GitHub: https://github.com/WebAssembly/WASI/blob/main/Proposals.md

A maioria das propostas de API do WASI começou originalmente com o uso de arquivos WITX. O formato WITX é considerado atualmente um formato legado pelos mantenedores do WASI, apesar de sua importância ser ainda bastante discutida dentro da comunidade. O formato WITX é usado para definir interfaces, adicionando o suporte para tipos de módulos e anotações.

Originalmente, o formato WITX foi inspirado no formato WIT, porém são formatos diferentes. O grupo responsável pelo WASI migrou o WASI Core (e continua migrando as demais propostas) do WITX para WIT, porque o WIT fornece melhor suporte para linguagens não semelhantes a C, melhor suporte para virtualização de APIs e suporta APIs mais expressivas.

O formato WIT faz parte da especificação atual do WASI. Hoje, as APIs estão em um processo de transição da reescrita do formato WITX para WIT e ABI canônica (veremos mais a respeito na próxima seção, sobre arquitetura). Para melhor visualização, veja como o formato WIT é estruturado no código a seguir:

```
// interface.wit
default world minha-interface {
  import log: func(msg: string)
```

```
  export executar: func()
}
```

Nesse código, na definição da interface, definimos a palavra `world`, que existe no contexto do WIT e especifica que a interface `minha-interface` será disponível no contexto de exportação e importação de um módulo WebAssembly.

Se o arquivo `interface.wit` for vinculado pelo compilador, o compilador criará a interface para o binário gerado com a saída em WASI. O formato WIT não tem a mesma relevância na saída `wasm32-unknown-unknown`. Existem ferramentas que fazem a vinculação automática do WIT para binários destinados em WASI, como o projeto WIT-bindgen (github.com/bytecodealliance/wit-bindgen).

A especificação do formato WIT é bem experimental e vem sofrendo mudanças ano após ano, então não vamos entrar em detalhes sobre como o WIT funciona devido à atual instabilidade. O que vale saber é que o WIT tem um papel importantíssimo na portabilidade de APIs de sistema para o WASI, permitindo que desenvolvedores definam suas próprias interfaces e as utilizem posteriormente. Para mais informações sobre o WIT, consulte a documentação no repositório oficial do WASI. O WIT é apenas uma das peças da vasta arquitetura do WASI. Na próxima seção, vamos clarificar como a arquitetura do WASI foi construída para trazer o WebAssembly para perto da camada do sistema.

11.2 Como a arquitetura do WASI funciona?

A arquitetura do WASI foi projetada para tornar módulos WebAssembly independentes de navegadores; logo, todos os padrões seguem a premissa de não depender de integrações de Web APIs ou JavaScript. Além da independência do ecossistema da Web, as especificações do WASI revisitam os conceitos de distribuição e o

acesso dos módulos WebAssembly, refletindo em questões de segurança.

A segurança do WASI é baseada em capacidade integrada, de modo que estende o *sandboxing* característico do WebAssembly para incluir I/O. ***Sandboxing*** (também conhecido como criação de *sandbox*, ou caixa de areia) é a prática de isolar um software para que ele possa acessar apenas determinados recursos, programas e arquivos dentro de um sistema de computador, a fim de reduzir o risco de erros ou programas maliciosos que afetam o sistema.

O design da segurança baseada em capacidade integrada do WASI foi inspirado pelos respectivos projetos: *CloudABI* (não é mantido; https://github.com/NuxiNL/cloudlibc) e *Capsicum* (https://www.cl.cam.ac.uk/research/security/capsicum/), que se encaixa bem no modelo *sandbox* do WebAssembly.

A segurança baseada em capacidade é um conceito no projeto de sistemas de computação seguros, um dos modelos de segurança existentes. Uma **capacidade** (também conhecida como *chave*) é um símbolo de autoridade comunicável e que não pode ser falsificado. Refere-se a um valor que faz referência a um objeto junto com um conjunto associado de direitos de acesso.

As capacidades são usadas por qualquer programa de usuário e refletem a permissão de acesso dos objetos. Um objeto pode ser arquivos, diretórios, soquetes de rede e outros recursos. Os objetos são identificados por descritores de arquivo semelhantes ao UNIX, que são índices em tabelas externas cujos elementos representam capacidades.

O funcionamento da segurança baseada em capacidade é muito semelhante ao do WebAssembly, o que foi um fator determinante para seu uso na arquitetura do WASI. O WASM não fornece capacidade de acessar o mundo externo sem a chamada de funções importadas. O WASI segue o mesmo princípio do WebAssembly: as

APIs não têm permissão para acessar o mundo externo sem um recurso associado.

Por baixo dos panos, o WASI possui uma infraestrutura preparada para lidar com capacidade. Por exemplo, em vez de realizar uma típica chamada de sistema aberto, o WASI fornece uma chamada de sistema similar ao `openat` do UNIX, que exige que o processo de chamada tenha um descritor de arquivo para um diretório que contém o arquivo. O descritor de arquivo identifica e representa a capacidade de abrir arquivos em um determinado diretório.

Além da chamada de sistema similar ao `openat`, o WASI provê uma implementação do `libc` (biblioteca padrão escrita em C) chamada `WASI libc`. O `WASI libc` está dentro de uma camada que prepara o `libc` para ambientes que são baseados em capacidade (de forma similar ao projeto github.com/musec/libpreopen).

A estrutura atual do WASI está sujeita a evoluções futuras. Frequentemente a arquitetura do WASI é discutida com a possibilidade de exploração de conceitos existentes, como as APIs de baixo nível do projeto Google Fuchsia (sistema operacional do Google baseado em capacidade), que são mais complexas e abstratas, embora também mais capazes.

> Se você quer aprender todos os detalhes do design da arquitetura do WASI, recomendo posteriormente acessar e estudar a documentação da arquitetura provida pelo repositório oficial do WASI: https://github.com/WebAssembly/WASI.

Aprendemos nesta seção como o WASI permite a execução e acesso das camadas de sistema. Todas as chamadas de sistema precisam ser feitas de forma declarativa, geralmente no lado de quem programa. Esta operação pode ser feita na escolha das APIs que serão utilizadas.

Em cada chamada de sistema definida pelo módulo WebAssembly, o WASI vai se encarregar de verificar se a capacidade provida para a API selecionada é de fato válida. Mas como de fato funciona a definição e uso das APIs do WASI no WebAssembly?

11.3 Como usar WebAssembly com WASI?

Há diferentes propósitos do uso do WASI. Algumas possibilidades de uso são:

- Aplicativos e jogos multiplataforma: você pode ter um único binário ou executável, que pode ser executado em qualquer plataforma que tenha um tempo de execução do WebAssembly;
- Reutilização de código entre plataformas e casos de uso: você pode imaginar a reutilização de código em toda a arquitetura de seu aplicativo entre diferentes plataformas;
- *Containers* (empacotamento de aplicações): contêineres são pacotes de software que contêm todos os elementos necessários para rodar em qualquer ambiente. O uso do WASI permite que aplicativos e suas dependências possam ser compilados para um ou mais WebAssembly que rodam em tempo de execução.

Muitas outras ideias podem se tornar possíveis com o uso de WebAssembly com WASI. Entretanto, a especificação está em fase de desenvolvimento e o hospedeiro (*host*, em inglês) usado precisa ter habilitado a implementação da especificação, do contrário, podem ocorrer cenários onde recursos necessários não estão disponíveis.

Os hospedeiros são ambientes de tempo de execução do WebAssembly que são responsáveis pela execução dos seus módulos WebAssembly. Durante o livro, usamos o navegador, logo o seu navegador é o ambiente de tempo de execução — por exemplo, no

caso do navegador Google Chrome, é o V8. Quando executamos o WebAssembly em um servidor, ao contrário do navegador, muitas vezes você não precisa do suporte para HTML, CSS, JavaScript e outros. Existem hospedeiros que foram desenhados e criados para serem executados no servidor (além do fato de serem mais leves que um navegador), como o *Wasmtime*, *Wasmer*, *Lucet*, *Wasm3* e outros.

Os hospedeiros podem receber convidados, que são os módulos WebAssembly executados pelo hospedeiro. Os hospedeiros são capazes de fornecer funcionalidades adicionais ao convidado. As funcionalidades são oferecidas passando funções e valores para o objeto de importações. Um hospedeiro que tem a implementação dos padrões do WASI permite realizar ações no nível do sistema para os convidados (módulos WebAssembly) e, dessa forma, desenvolvedores podem escrever módulos WASM que podem acessar os recursos do sistema!

Usaremos o **Wasmtime** como a opção de hospedeiro neste capítulo. O principal motivo da escolha é o fato de que a maioria dos membros responsáveis pelas especificações do WASI faz parte do grupo da Bytecode Alliance, que mantém o Wasmtime, logo, o tempo de implementação de padrões do Wasmtime geralmente fica próximo à vanguarda do WASI.

11.4 Wasmtime

> **Lembrete:** Se você tiver curiosidade ou precisar verificar algo, todos os exemplos do livro (incluindo os arquivos binários) estão neste link: https://github.com/raphamorim/livro-webassembly-exemplos. Basta clonar o repositório usando git ou fazer uso da opção de download do código-fonte que o GitHub provê.

O Wasmtime é um hospedeiro multiplataforma para a execução de programas WebAssembly. Foi desenvolvido com base no gerador de código chamado *Cranelift*, que é otimizado para gerar rapidamente código de máquina de alta qualidade em tempo de execução ou antecipadamente. Seu tempo de execução também é otimizado para casos comuns de servidores *back-end*, como instanciamento eficiente e rápido, escalabilidade de instâncias concorrentes e transições de baixa sobrecarga (termo originário do inglês *low-overhead*) entre o hospedeiro e o convidado.

O desenvolvimento do Wasmtime é regrado em estar alinhado com os testes oficiais do WebAssembly. O Wasmtime também é responsável por implementar a API C oficial do WASM e, como dito anteriormente, os desenvolvedores do Wasmtime estão diretamente envolvidos com o processo de padrões do WebAssembly ao longo do caminho.

O Wasmtime pode ser instalado no Linux ou macOS a partir do seguinte script de instalação na linha de comando:

```
curl https://wasmtime.dev/install.sh -sSf | bash
```

Se você possui um sistema operacional Windows, é necessário baixar os instaladores e binários diretamente da página de *releases*: github.com/bytecodealliance/wasmtime/releases.

Após a instalação do Wasmtime, você pode executar qualquer arquivo binário do WebAssembly usando o seguinte comando:

```
wasmtime caminho-do-arquivo-binário.wasm
```

Antes de sairmos executando arquivos binários do WebAssembly como se não houvesse amanhã, que tal colocarmos um objetivo curto e prático? Considere o seguinte cenário: queremos criar um módulo que, quando executado, mostra o valor "85" do `stdout` (do inglês, *Standard output*, que significa "saída padrão") na linha de comando. Até então, construímos o formato de biblioteca no

decorrer do livro, mas, para binários executáveis, precisamos definir uma função para ser executada automaticamente.

Vimos no capítulo 9 que é possível fazer isso com WebAssembly com o uso da instrução `start`. Uma função pode ser associada à instrução `start` e ser executada assim que o módulo é instanciado. A tentativa lógica para resolução do problema seria escrever uma função que retorna 85 e atribuir a ela a instrução `start`, como no código a seguir:

funcao-retorna-85.wat

```
(module
  (func $retorna85 (result i32)
  i32.const 85
  return)
  (start $retorna85)
)
```

No entanto, temos dois problemas com essa abordagem: o primeiro é que o código escrito não é válido para compilação. Funções associadas à instrução `start` não podem ter qualquer tipo de retorno. O segundo problema é que retornar um valor não é necessariamente suficiente para imprimir o resultado. Quando executamos uma função do WebAssembly no navegador, estamos fazendo a impressão usando o `console.log` do JavaScript.

Atualmente, o WASI permite fazer impressões de retorno de valores com um parâmetro chamado `--invoke`. Esse parâmetro recebe o nome da função e também pode receber os valores que serão enviados para a função. Se reescrevermos o código do `funcao-retorna-85.wat` para a versão do código a seguir:

```
(module
  (func (export "retorna85") (result i32)
  i32.const 85
  return)
)
```

Será possível executar o comando e obter o valor "85". Todavia, invocações de funções com retorno é algo bem experimental no Wasmtime e não é certo que vá funcionar no futuro. Assim que você executar o comando, será retornada uma mensagem de alerta, como no exemplo a seguir:

```
$ wasmtime funcao-retorna-85.wasm --invoke retorna85
warning: using `--invoke` with a function that returns values is
experimental and may break in the future
85
```

A parte mais interessante (e talvez mais difícil) do WASI é mudar a mentalidade de como vemos o WebAssembly, frequentemente visto como uma biblioteca em vez de uma aplicação executável. O exemplo que acabamos de ver não usa nenhum recurso do WASI: não estamos fazendo nenhuma operação como o `console.log` do JS ou `println!` do Rust.

Se você quiser ter acesso às APIs do WASI, apenas executar um binário do WebAssembly (feito para ser executado no navegador) não será suficiente para usar qualquer funcionalidade do WASI. Lembra-se de que eu falei que o hospedeiro provê ferramentas adicionais que são passadas para a definição de importações? Pois bem, precisamos declarativamente especificar que queremos usar as APIs do WASI e isso significa usar as ferramentas que o WASI provê. Vamos visualizar isso de forma prática.

Nas próximas seções, veremos como o WASI funciona com Rust e WAT, assim podemos ver as diferenças entre as duas abordagens. Primeiro vamos começar vendo como o WASI funciona com Rust.

11.5 WASI com Rust

Para usar WASI com Rust, é preciso instalar outro destino de compilação conhecido como `wasm32-wasi`. O `wasm32-wasi` é integrado

automaticamente à biblioteca padrão do Rust e destina-se à produção de binários autônomos.

A biblioteca padrão do Rust no `wasm32-wasi` funciona de forma diferente do destino `wasm32-unknown-unknown`, onde há diversas inconsistências, como módulos que falham automaticamente quando usados ou funções que retornam um endereço de memória em vez do valor esperado. No `wasm32-wasi`, temos a biblioteca padrão completamente integrada para uso.

A instalação pode ser feita através do gerenciador oficial de ferramentas do Rust, conhecido como `rustup`. Execute o comando a seguir para proceder com a instalação do `wasm32-wasi`:

```
rustup target add wasm32-wasi
```

Pegamos parte do problema anterior sobre mostrar 85 no `stdout` no terminal e também adicionaremos o retorno de uma *string* aleatória. Crie o arquivo `wasi-com-rust.rs` e preencha-o com o código a seguir:

```
fn main() {
  println!("Bem-vindo ao fantástico mundo do WebAssembly.");
  println!("{}", 85);
}
```

Depois de criar o arquivo `wasi-com-rust.rs`, faça a compilação com o compilador do Rust, o `rustc`, passando o parâmetro de destino especificado em `wasm32-wasi`:

```
rustc wasi-com-rust.rs --target wasm32-wasi
```

Esse comando criará um arquivo do binário do WebAssembly. Há uma grande diferença do conteúdo do binário criado agora e o conteúdo do binário criado com `wasm32-unknown-unknown`. Se você executar qualquer ferramenta de depuração no binário `wasi-com-rust.wasm`, notará que o arquivo contém o conteúdo não só das importações das APIs do WASI, mas também funções auxiliares que foram incluídas pelo compilador do Rust.

Agora que temos o arquivo `wasi-com-rust.wasm`, podemos finalmente executar o Wasmtime e obter os resultados que esperamos:

```
$ wasmtime wasi-com-rust.wasm
Bem-vindo ao fantástico mundo do WebAssembly.
85
```

Se considerarmos que, no WASI, podemos usar a biblioteca padrão do Rust sem limitações (ou melhor dizendo, inconsistências), é certo assumir que a velocidade de desenvolvimento vai se beneficiar disso. Para poder explorar um pouco mais a biblioteca padrão do Rust no WASI, vamos criar outro arquivo e adicionar alguns exemplos.

Lembra que, para trabalhar com *strings* diretamente no `wasm32-unknown-unknown`, precisamos aplicar a codificação de dados? Pois bem, no `wasm32-wasi`, podemos usar os tipos providos pelo Rust diretamente sem nos preocuparmos com a codificação e decodificação de dados, por exemplo, o uso do `&str`.

Que tal vermos como usar *strings* no WASI na prática? Para isso, vamos escrever um validador de anagramas (e, de quebra, ainda veremos o funcionamento de vetores).

Um anagrama é uma palavra ou frase formada pela reorganização das letras de uma palavra ou frase diferente, geralmente usando todas as letras originais exatamente uma vez. Uma função que valida anagramas recebe duas listas de caracteres e faz a comparação entre ambas.

Criaremos o arquivo `wasi-anagrama.rs` e adicionaremos nossa função de validação de anagrama chamada `validar_anagrama`:

```
#[no_mangle]
fn validar_anagrama(s: &str, t: &str) -> bool {
  let mut s = s.to_ascii_lowercase().chars().collect::<Vec<_>>();
  let mut t = t.to_ascii_lowercase().chars().collect::<Vec<_>>();
  s.sort_unstable();
  t.sort_unstable();
  s == t
```

```
}

fn main() {
  println!("{}", validar_anagrama("amor", "ramo"));
  println!("{}", validar_anagrama("mora", "roma"));
  println!("{}", validar_anagrama("marcia", "paulo"));
}
```

Compile o código com o compilador do Rust, o `rustc`, habilitando o destino de saída como `wasm32-wasi`:

```
rustc wasi-anagrama.rs --target wasm32-wasi
```

Um novo arquivo binário será criado e, quando executado, retornará os seguintes resultados:

```
$ wasmtime wasi-anagrama.wasm
true
true
false
```

Bem legal! É notável que usar a biblioteca padrão do Rust agiliza o desenvolvimento.

Perceba também que o formato do módulo é diferente das bibliotecas que escrevemos no decorrer do livro, com retorno de ponteiros juntamente com a ABI do C, pois segue o formato de executável em vez do formato de biblioteca. Um formato executável geralmente possui nenhuma ou consideravelmente menos funções expostas do que um formato de biblioteca.

Além de podermos usar os tipos da biblioteca padrão do Rust, como vetores, *strings*, inteiros e outros, podemos usar as APIs que são mapeadas para uso com WASI, por exemplo, a API de arquivo.

11.6 Escrevendo em arquivo com WASI

Vamos escrever outra aplicação para ver o uso da API de arquivo em prática. Neste exemplo adicionaremos também a resolução de um problema matemático usando inteiros sem sinal de 64 bits. O algoritmo que resolveremos é conhecido como *soma alíquota* (o termo em inglês é *aliquot sum*).

A soma alíquota é a soma de um valor inteiro positivo (nomeareamos o valor como *x*) com todos os divisores próprios de seu respectivo valor, exceto o próprio *x*. Um exemplo de soma alíquota: dado o valor 15, os divisores próprios de 15 (divisores positivos que não são iguais a 15) são 1, 3 e 5, então a soma alíquota de 15 é 9 (1 + 3 + 5).

Os diferentes valores usados na função `soma_aliquota` deverão ser salvos em um arquivo chamado `arquivo.txt` dentro da pasta `resultado`, mas não se preocupe se o caminho da pasta não existir agora; por ora, vamos criar o arquivo chamado `wasi-arquivo-escrita.rs` com o respectivo código:

```
use std::fs::File;
use std::io::prelude::*;

#[no_mangle]
pub extern fn soma_aliquota(numero: u64) -> u64 {
  if numero == 1 || numero == 0 {
      return 0;
  }
  let mut soma: u64 = 0;
  for i in 1..(numero / 2 + 1) {
      if numero % i == 0 {
          soma += i;
      }
  }
  soma
}

fn main() -> std::io::Result<()> {
  let mut file = File::create("/resultado/arquivo.txt")?;
```

```
  file.write_all(b"Resultado da soma_aliquota de 15: ")?;
  file.write_all(soma_aliquota(15).to_string().as_bytes())?;
  file.write_all(b"\n")?;
  file.write_all(b"Resultado da soma_aliquota de 85: ")?;
  file.write_all(soma_aliquota(85).to_string().as_bytes())?;
  Ok(())
}
```

Existem formas mais otimizadas de fazer a escrita desses dados, porém optei pela forma mais didática, que é considerando a entrada constante de *strings*. Compile o arquivo `wasi-arquivo-escrita.rs` para o binário do WebAssembly com o destino de compilação em WASI:

```
rustc wasi-arquivo-escrita.rs --target wasm32-wasi
```

Se executarmos o módulo com o Wasmtime, obteremos um erro bastante interessante: o caminho da pasta `/resultado` não existe.

```
wasmtime wasi-arquivo-escrita.wasm
Error: Custom { kind: Uncategorized, error: "failed to find a pre-opened
file descriptor through which \"/resultado/arquivo.txt\" could be opened"
}
```

O erro vai continuar mesmo se o caminho da pasta `/resultado` existir no seu sistema operacional. Isso porque o Wasmtime precisa que você mapeie as pastas e arquivos que estão sendo usados, do contrário o hospedeiro não vai procurar pelo caminho no formato do seu sistema operacional. Você pode mapear um caminho de pasta usando o parâmetro `mapdir`, que recebe uma *string* padronizada no formato `caminho-do-convidado::caminho-do-hospedeiro`.

Como o caminho da pasta definido no convidado é `/resultado/`, podemos simplesmente fazer o caminho do hospedeiro ser um `.` (ponto final) declarativamente, e isso será suficiente para criar o arquivo no mesmo diretório em que estivermos executando o comando. Siga o comando:

```
wasmtime wasi-arquivo-escrita.wasm --mapdir /resultado/::.
```

Um novo arquivo chamado `arquivo.txt` será criado após a execução desse comando. Você pode checar o conteúdo do arquivo criado de diferentes formas (exemplo: `cat arquivo.txt`) e deverá ser igual ao seguinte conteúdo:

```
Resultado da soma_aliquota de 15: 9
Resultado da soma_aliquota de 85: 23%
```

Com esse exemplo, encerramos a parte de WASI com Rust, mas recomendo fortemente continuar a explorar outras APIs da biblioteca padrão do Rust que foram traduzidas para o WASI. Existem inúmeras outras possibilidades como criptografia de dados, codificadores de vídeos, e tantas outras aplicações WebAssembly que foram escritas com Rust e WASI. Uma combinação realmente muito poderosa.

Na próxima seção, veremos como o WASI funciona no formato de texto do WebAssembly, o WAT, e vamos entender melhor como um módulo WebAssembly usa os recursos de um hospedeiro.

11.7 WASI com WebAssembly Text Format

Você pode criar aplicações WASI usando o formato de texto do WebAssembly da mesma forma que criamos binários do WAT, usando ferramentas como o pacote oficial de ferramentas binárias (WABT), Binaryen, `wasm-tools` e outros. Neste sentido, a forma como criamos binários para usar recursos do WASI é diferente do Rust, pois não é preciso adicionar mais um destino de compilação.

Isso se deve ao fato de que o compilador do Rust, quando compila para `wasm32-unknown-unknown` , está ignorando a biblioteca padrão. No caso do `wasm32-wasi` , o destino de compilação provê a biblioteca padrão mapeada para uso do WASI. Com base nessa premissa, é correto assumir que o formato WAT que vamos escrever será similar a usar `wasm32-unknown-unknown` como destino de compilação, ou seja,

declarativamente mapeamos o que desejamos importar e usar do WASI.

Vamos ver isso de forma prática quando escrevermos o módulo WebAssembly que usa funções do WASI. Para isso, criaremos um arquivo nomeado como `wasi-com-wat.wat` que contém o menor módulo possível:

```
(module)
```

Neste módulo que acabamos de criar, vamos fazer uma operação de saída de dados na linha de comando do terminal (`stdout`) usando uma função da especificação do WASI provida pelo nosso hospedeiro, o Wasmtime.

Atualmente, a maioria das funções da especificação do WASI segue o padrão `wasi_unstable` e o nome da função, com níveis de aninhamento para importação. A nomenclatura `wasi_unstable` é o padrão agora, mas já foi `wasi_snapshot_preview1` e poderá sofrer mudanças assim que a API for normalizada. Todas as importações de um módulo WebAssembly precisam necessariamente acontecer no topo do arquivo.

Importaremos a função `fd_write` do `wasi_unstable` , que escreverá os vetores I/O fornecidos no `stdout` . Os vetores I/O fazem parte do conceito de I/O vetorizado, que é um método de entrada e saída pelo qual um processo lê sequencialmente os dados de vários *buffers* e os grava em um único fluxo de dados, ou lê dados de um fluxo de dados e os grava em vários *buffers*.

A função `fd_write` se baseia em quatro parâmetros:

- `$fd` : representa o descritor de arquivo usado, o valor 1 aponta para o `stdout` ;
- `$iovs` : o ponteiro para o I/O vetorizado (em inglês, *array iov*);
- `$iovs_len` : quantidade de itens do I/O vetorizado;
- `$nwritten` : um lugar na memória para armazenar o número de bytes escritos.

O retorno da função é o número de bytes gravados:

```
(func $fd_write (import "wasi_unstable" "fd_write")
  (param $fd i32)
  (param $iovs i32)
  (param $iovs_len i32)
  (param $nwritten i32)
  (result i32))
```

A função `$fd_write` é apenas uma das funções à sua disposição. Você pode ver a lista completa de funções que o Wasmtime disponibiliza na documentação oficial: https://docs.wasmtime.dev/api/wasi_common.

Após a adição da importação da função `fd_write`, podemos seguir em frente com nosso código WAT e adicionar a definição de memória junto aos respectivos dados da memória:

```
(memory 1)
(export "memory" (memory 0))

(data (i32.const 8) "> Olá!\n")
(data (i32.const 16) "> Está tudo bem?\n")
```

Nesse código, configuramos a memória com o tamanho de uma página (64 *kibibytes*) e definimos dois blocos de dados, que são *strings* (`"> Olá!\n"` e `"> Está tudo bem?\n"`). Até então, todos os arquivos WAT que escrevemos seguiam o formato de biblioteca, mas queremos um formato executável desta vez (assim como usamos a função `main` nos exemplos com Rust). É bem comum tentar reproduzir a característica de autoexecutável da função `main` do Rust com o uso da instrução `start`.

A instrução `start` não funcionará para o nosso caso, pois ela executa a função sem o devido acesso à lista de exportações (por exemplo, a memória). A instrução `start` foi criada para ser um auxiliar no instanciamento do módulo, mas é limitada a diversos cenários. Em razão desse problema, foi criado o padrão de execução automática de qualquer função nomeada como `_start`.

No código a seguir, estamos definindo a função `$main` e exportando como `_start`. Note que também criamos o vetor I/O e passamos para a função `$fd_write`.

```
(func $main (export "_start")

  ;; Criando um novo vetor I/O na memória linear
  (i32.store (i32.const 0) (i32.const 8))
  (i32.store (i32.const 4) (i32.const 26))

  (call $fd_write
      ;; Descritor de arquivo, valor 1 para stdout.
      (i32.const 1)

      ;; Ponteiro para o array iov, que é armazenado na posição 0 da
memória.
      (i32.const 0)

      ;; Mesmo que tenhamos dois blocos de dados, ainda estamos alocando
tudo em 1 string.
      (i32.const 1)

      ;; O lugar na memória para armazenar o número de bytes escritos.
      (i32.const 28)
  )
  ;; Descarta o número de bytes gravados no topo da pilha
  drop
)
```

Note que estamos propositalmente descartando os bytes gravados no topo da pilha (valor que retorna após a execução do `$fd_write`), pois a função `$main` especifica que não deve haver retorno. Para descartar o valor da pilha, usamos a instrução `drop`, do contrário a função retornaria um `i32` e teríamos um erro de compilação.

Se você tiver interesse, é possível ler toda a discussão sobre o uso de `start` como instrução e funções nomeadas como `_start` no repositório oficial no GitHub: https://github.com/WebAssembly/WASI/issues/19.

Após a adição da função anterior, podemos concluir nosso executável que escreve no `stdout` e fazer sua respectiva compilação com o comando `wat2wasm wasi-com-wat.wat`. O código completo do módulo deverá ser igual ao seguinte código:

```
(module
  (func $fd_write (import "wasi_unstable" "fd_write")
      (param $fd i32)
      (param $iovs i32)
      (param $iovs_len i32)
      (param $nwritten i32)
      (result i32))

  (memory 1)
  (export "memory" (memory 0))

  (data (i32.const 8) "> Olá!\n")
  (data (i32.const 16) "> Está tudo bem?\n")

  (func $main (export "_start")

      (i32.store (i32.const 0) (i32.const 8))
      (i32.store (i32.const 4) (i32.const 26))

      (call $fd_write
          (i32.const 1)
          (i32.const 0)
          (i32.const 1)
          (i32.const 4)
      )
      drop
```

```
  )
)
```

Após a compilação desse código para um arquivo binário WebAssembly, faça a execução do binário com o Wasmtime e você deverá ter o seguinte retorno:

```
$ wasmtime wasi-log.wasm
> Olá!
> Está tudo bem?
```

Fantástico, acabamos de escrever nosso primeiro executável WASI com WAT! Um dos benefícios de criar executáveis WASI com WAT é que o formato de texto do WebAssembly geralmente está mais atualizado em termos de suporte de funcionalidades quando comparado com os compiladores de outras linguagens. Muitas vezes você precisa esperar lançamentos de versões de um compilador ou ferramenta específica para poder ter novos recursos do WASI no destino de compilação.

Um exemplo atual do cenário descrito é a compatibilidade da API de *pthreads* do WASI, que ainda não é suportada no Rust com a biblioteca padrão (`wasm32-wasi`), mas funciona no Wasmtime. Se a API está disponível para uso no hospedeiro, você pode fazer uso direto com o WAT.

> Nota: dependendo do caso, é também possível usar APIs que não têm suporte no `wasm32-wasm` por meio da compilação com `wasm32-unknown-unknown` e do uso de vinculação declarativa dos métodos.

Dito isso, vamos para a última seção do livro, que aborda como a API de *pthreads* do WASI funciona.

11.8 Usando pthreads com WASI

No capítulo 7 deste livro, vimos sobre o uso de instruções atômicas e memória compartilhada no WebAssembly, que permitem criar o sistema de *threads*. O WebAssembly provê tais recursos, porém são necessárias camadas adicionais de controle e criação de *threads*. No exemplo dado no capítulo 7, usamos *Web Workers* para acessar a memória compartilhada e executar instruções atômicas.

O WASI é um passo à frente das primitivas que o WebAssembly provê. A proposta do WASI disponibiliza apenas um mecanismo para geração de *threads*: o uso de *pthreads*. Na computação, ***pthreads*** (também conhecidas como *POSIX Threads*) são um modelo de execução que existe independentemente de uma linguagem de programação, bem como um modelo de execução paralela.

A *pthread* permite que um programa controle vários fluxos de trabalho diferentes que se sobrepõem no tempo. Cada fluxo de trabalho é referido como um encadeamento, e a criação e o controle sobre esses fluxos são obtidos por meio de chamadas à API da *pthread* provida pelo Unix (lembrando que POSIX é a interface de sistema operacional portátil do Unix).

Assim como em todas as seções deste capítulo, vamos escrever um exemplo para ver como a API funciona de forma prática. Nosso objetivo é criar um executável WASI que cria *pthreads* e exibe um valor no `stdout`. Começaremos com a estrutura do módulo. Crie um arquivo com o formato de texto do WebAssembly nomeado como `wasi-threads.wat` contendo o seguinte código:

```
(module
  (func $fd_write (import "wasi_unstable" "fd_write")
      (param $fd i32)
      (param $iovs i32)
      (param $iovs_len i32)
      (param $nwritten i32)
      (result i32))
```

```
  ;; 64 KiB
  (memory 1)
  (export "memory" (memory 0))
```

Adicionamos a função `$fd_write` que usamos na seção anterior e definimos a memória com o tamanho de 64 KiB. Na última seção, usamos a função diretamente `$fd_write`, mas se quisermos fazer a exibição de diferentes *strings* no `stdout` várias vezes, isso vai afetar a legibilidade do código. No código a seguir, encapsulamos a função `$fd_write` em uma função auxiliar que recebe o ponteiro de onde a *string* foi salva e chama outra função, que retorna o tamanho automático da *string*, assim não precisamos escrever o tamanho declarativamente quando executarmos `$fd_write`:

```
(func $exibir
  (param $ponteiro i32)
  (i32.store
      (i32.const 0)
      (local.get $ponteiro))
  (i32.store
      (i32.const 4)
      (call $tamanhoDaString
          (local.get $ponteiro)))
  (call $fd_write
      (i32.const 1)
      (i32.const 0)
      (i32.const 1)
      (i32.const 3000))
  drop)
```

A lógica da função `$tamanhoDaString` é percorrer os dados de um dado ponteiro até achar um valor igual a `\0` e, até encontrar esse valor, ela vai incrementar a variável `$tamanho` em 1. O laço de repetição vai procurar pelo valor `\0` e, assim que o encontrar, vai terminar o laço e retornar o valor da variável `$tamanho`.

```
(func $tamanhoDaString (param $ponteiro i32) (result i32)
  ;; Define a variável $tamanho e colocar o valor como 0
```

```
  (local $tamanho i32)
  (local.set $tamanho
      (i32.const 0))
  ;; Procura pelo final da string, o valor: '\0'.
  ;; Continua adicionando 1 até achar.
  (block $procuraTamanho
      (loop $repeticao
          (br_if $procuraTamanho
              (i32.eqz
                  (i32.load8_u
                      (i32.add
                          (local.get $ponteiro)
                          (local.get $tamanho)))))
          (local.set $tamanho
              (i32.add
                  (local.get $tamanho)
                  (i32.const 1)))
          (br $repeticao)))
  ;; Retorna o tamanho
  (local.get $tamanho))
```

Agora temos uma função que vai automaticamente procurar o tamanho da *string*, porém, ao mesmo tempo, adicionamos a dependência de os valores terminarem com `\0`. No código a seguir, descreveremos os dados das *strings* com o final `\0` e uma função `_start`, que é executada automaticamente pelo hospedeiro.

```
(data (i32.const 1000) "*\n\00")
(data (i32.const 1005) "1\n\00")
(data (i32.const 1010) "2\n\00")
(data (i32.const 1015) "3\n\00")
(data (i32.const 1020) "4\n\00")

(func (export "_start")
  (call $exibir
      (i32.const 1000))
  (call $exibir
      (i32.const 1005))
  (call $exibir
```

```
      (i32.const 1010))
  (call $exibir
      (i32.const 1015))
  (call $exibir
      (i32.const 1020))
)
```

Depois de adicionar esse código, podemos compilar nosso arquivo de texto para um binário executável com o `wat2wasm` e, depois, fazer a execução do binário com o Wasmtime.

```
$ wat2wasm wasi-threads.wat
$ wasmtime run ./wasi-sem-threads.wasm
*
1
2
3
4
```

Até então, não temos nenhuma *pthread* sendo criada, e o mesmo executável vai produzir o mesmo resultado para cada execução. A partir de agora, adicionaremos o suporte a *pthreads* no nosso executável e usaremos os dados das *strings* existentes, que terão diferentes representações:

- O símbolo `*` representará o processo principal;
- Os números representarão as *pthreads* de acordo com sua ordem de lançamento.

Como vimos no capítulo sobre as instruções atômicas, a memória necessariamente precisa ser configurada como compartilhada (além de possuir operações de controle de volume da memória, conhecida como operações *bulk*), do contrário, não será possível usar as primitivas atômicas. No código a seguir, configuramos a memória como compartilhada:

```
;; 64Kib
(memory $memoria (import "env" "memory") 1 1 shared)
(export "memory" (memory $memoria))
```

A próxima etapa é usar a função provida pelo Wasmtime baseada na especificação do WASI conhecida como `thread-spawn`. A função `thread-spawn` recebe um parâmetro `i32` que você pode usar à vontade, e o retorno também é em `i32`.

O valor do retorno da função `thread-spawn` é baseado no processo de criação da *thread*, se o processo foi bem-sucedido o valor é positivo e representa um identificador exclusivo. Se o processo não foi bem-sucedido, o valor é negativo e representa o código de erro.

```
(func $gerar_thread (import "wasi" "thread-spawn")
  (param $arg i32)
  (result i32))
```

Repare também que a função `thread-spawn` está no nível `wasi` em vez de `wasi_unstable` (como a função `fd_write`). Isso se deve ao fato de que a API de *threads* está em nível experimental em vez de estar agrupada em `wasi_unstable`, e é necessário especificar o uso da API de *threads* via linha de comando.

Agora que temos o poder de criar *pthreads* na nossa mão, podemos atualizar nossa função `_start` para criar uma *pthread* para cada ponteiro de dados numéricos de *string* (1, 2, 3 e 4):

```
(func (export "_start")
  ;; Exibe "*"
  (call $exibir
      (i32.const 1000))

  (drop (call $gerar_thread
      (i32.const 1005)))
  (drop (call $gerar_thread
      (i32.const 1010)))
  (drop (call $gerar_thread
      (i32.const 1015)))
  (drop (call $gerar_thread
      (i32.const 1020)))
)
```

A função `gerar_thread` recebe como argumento a posição inicial da *string* na memória e, a cada vez que a função é chamada, o hospedeiro lançará a execução de uma *pthread*. Entretanto, da forma como está, isso não fará nada além de criar *pthread*. A proposta de *threads* do WASI define que uma função exportada como `wasi_thread_start` seja executada na criação da *pthread*.

A função `wasi_thread_start` recebe o identificador da *pthread* e o argumento que passamos quando criamos a *pthread*. No código a seguir, estamos declarando que, para cada vez que uma *pthread* for criada, devem ser impressos os dados da *string* com base no ponteiro recebido:

```
(func (export "wasi_thread_start")
  (param $id i32)
  (param $arg i32)
  (call $exibir
      (local.get $arg))
)
```

Se compilarmos o arquivo de texto da mesma forma que no exemplo de escrita em arquivo da seção anterior, vamos obter primeiramente um erro de compilação, pois a memória compartilhada só pode ser compilada pelo `wat2wasm` com a funcionalidade de *threads* habilitada: `--enable-threads`. É necessário dizer ao hospedeiro que a funcionalidade está habilitada usando `--wasm-features threads` no momento da execução do binário.

Assim que a compilação for executada com sucesso e desejarmos fazer a execução do módulo, vamos obter um erro no instanciamento do módulo, pois a função `thread-spawn` é inacessível no nível de importações conhecido como `wasi` sem o uso da opção que habilita a API de *threads*: `--wasi-modules experimental-wasi-threads`.

Agora que sabemos disso, utilizaremos os comandos de compilação (`wat2wasm`) e execução (`wasmtime`) para fazer uso dos parâmetros

corretos. A seguir, temos um exemplo de compilação e execução com o uso de *pthreads*:

```
$ wat2wasm wasi-threads.wat --enable-threads
$ wasmtime run --wasm-features threads --wasi-modules experimental-wasi-threads ./wasi-threads.wasm
*
3
2
1
4
```

É incrível ver que temos o poder de recurso nativo no WebAssembly. A ordem de processamento das *threads* não é síncrona, logo, a cada vez que você fizer a execução, obterá resultados diferentes. Veja alguns exemplos de resultados de execução:

Exemplo 1

```
$ wasmtime run --wasm-features threads --wasi-modules experimental-wasi-threads ./wasi-threads.wasm
*
1
2
```

Exemplo 2

```
$ wasmtime run --wasm-features threads --wasi-modules experimental-wasi-threads ./wasi-threads.wasm
*
2
3
1
```

O nosso programa não tem nenhum tipo de gerenciamento da execução das *pthreads* criadas, ou seja, não esperamos nenhuma *pthread* terminar sua respectiva execução; logo, o programa pode encerrar antes mesmo até de uma ou mais *pthreads* imprimirem os dados.

Para aplicar níveis de controle em *pthreads*, podemos usar as instruções atômicas. Por exemplo, se quisermos garantir que a *pthread* com o valor 1 imprima os dados no `stdout` antes de fecharmos o processo, podemos escrever uma função chamada `$esperar` que tem um laço de repetição para ver se o valor da *pthread* foi atualizado.

A seguir, temos o código da função `$esperar`:

```
(func $esperar
  (param $endereco i32)
  (param $valorEsperado i32)
  (block $terminar (loop $repetir
      ;; se a operação atômica wait32 retornar
      ;; o valor 0, significa que o valor dentro
      ;; do endereço da memória especificado é
      ;; o valor esperado.
      (i32.eqz
          (memory.atomic.wait32
          ;; endereço da memória
          (local.get $endereco)
          ;; o valor esperado
          (local.get $valorEsperado)
          ;; -1 representa timeout infinito
          (i64.const -1)))
          (if
              (then (br $terminar))
              (else (br $repetir)))))
)

(func (export "_start")
  ;; imprime "*""
  (call $exibir
      (i32.const 1000))

  (drop (call $gerar_thread
      (i32.const 1005)))
  (call $esperar
      (i32.const 2000) (i32.const 1))
```

```
  ;; ...
```

A instrução `memory.atomic.wait32` espera por um valor especificado em um determinado endereço de memória. Você pode configurar um tempo limite (*timeout*) para a operação. No nosso código, usamos o laço de repetição para bloquear o processo principal, mas existem outras formas mais elegantes de fazer isso.

Adicionamos também a chamada da função `$esperar` dentro da função `_start`, após a geração da primeira *pthread*. Agora a função `$esperar` vai bloquear a continuação do resto da função `_start` até a validação do laço de repetição da função `$esperar` ser resolvida.

Só que a função `$esperar` espera pelo endereço de memória ser igual ao valor especificado; logo, precisamos alterar o valor do endereço de memória. Atualizaremos nossa função `wasi_thread_start` para adicionar a instrução `i32.atomic.rmw.add`, que incrementa o valor 1 no endereço "2000" (o mesmo endereço que estamos monitorando por mudanças):

```
(func (export "wasi_thread_start")
  (param $thread_id i32)
  (param $arg i32)

  (call $exibir
      (local.get $arg))
  (drop
      (i32.atomic.rmw.add
          ;; Endereço de memória
          (i32.const 2000)
          ;; Valor a ser adicionado
          (i32.const 1)))
  (drop
      (memory.atomic.notify
          ;; Endereço de memória
          (i32.const 2000)
          ;; Número de "ouvintes" para notificar
```

```
          (i32.const 1)))
)
```

A cada vez que o valor do endereço de memória é atualizado, realizamos também a operação `memory.atomic.notify`, que notifica um determinado ouvinte (processo que espera notificações) ou grupo de ouvintes avisando que o valor foi atualizado.

Se compilarmos e executarmos o executável agora, a primeira *pthread* sempre será exibida no `stdout` juntamente com o símbolo `*`. Confira o exemplo:

```
$ wat2wasm wasi-threads.wat --enable-threads
$ wasmtime run --wasm-features threads --wasi-modules experimental-wasi-threads ./wasi-threads.wasm
*
1
2
4
```

A forma como implementamos o controle das *pthreads* é efetiva para o nosso caso, onde estamos esperando apenas uma *thread* de forma bloqueante e síncrona; porém, usar laços de repetições para monitoramento de valores não seria a melhor solução se quiséssemos uma abordagem não bloqueante ou monitorar mais de um endereço de memória. Há outras formas de aplicar monitoramento não bloqueante, como usar as inúmeras instruções atômicas que auxiliam no controle (você pode ver a lista completa no GitHub, no seguinte endereço: https://github.com/WebAssembly/threads/blob/main/proposals/threads/Overview.md).

No decorrer deste capítulo, vimos a implementação de uso do WASI com Rust e com o WAT. Você tem a liberdade para escolher qual opção se encaixa melhor no caso de uso da sua aplicação, seja com uma linguagem de programação (como Rust, C ou outra) ou usando a representação textual do WebAssembly. O WebAssembly não nos prende a nenhuma linguagem. Com isso, estamos proficientes com

WASI e podemos escrever programas WebAssembly que podem ser executados em qualquer plataforma que tenha um tempo de execução do WebAssembly.

11.9 Conclusão

Você chegou ao final do livro! Bom trabalho! Quero agradecer a você por se juntar a esta jornada de aprendizado sobre o fantástico WebAssembly.

Aprendemos todos os fundamentos para escrever aplicações destinadas à Web sem precisar usar qualquer tipo de ferramenta, percorrendo conceitos como codificação de dados, serialização de tipos, vinculação dinâmica — conhecimento que permite até mesmo escrever um compilador próprio para WebAssembly, já que vimos como o formato de texto e estrutura binária funcionam. Neste exato momento, você está pronta(o) para implementar seus próprios projetos com WASM.

Recomendo dois próximos passos após a leitura deste livro: o primeiro é o engajamento com a comunidade, que é completamente opcional, mas bastante recompensador. Fazer parte da comunidade já nos coloca em uma posição privilegiada de informações sobre o estado atual e o futuro do WASM. Se você deseja engajar com a comunidade de outras programadoras e programadores que também são entusiastas do WebAssembly, existe uma lista de canais disponíveis no site oficial: https://webassembly.org/community/resources/.

O segundo passo após a leitura deste exemplar é dar asas à sua criatividade. Transformar uma ideia em um projeto prático é uma excelente forma de solidificar seu conhecimento. Se você estiver procurando inspirações, existe um website com uma lista de projetos de código aberto acessível: https://madewithwebassembly.com.

Esses próximos passos vão auxiliar você a se manter atualizado(a), pois o WebAssembly segue em constante evolução. Existem inúmeros outros recursos e funcionalidades que estão sendo discutidos e implementados como testes e experimentos na plataforma. Um mundo de possibilidades nos aguarda, seja para soluções multiplataforma com WebAssembly, quanto para uma Web mais rápida e eficiente.

Capítulo 12
Referências bibliográficas

ADOBE. *Adobe® RGB (1998) Color Image Encoding*. 2005. Disponível em: https://www.adobe.com/digitalimag/pdfs/AdobeRGB1998.pdf. Acesso em: 9 set. 2022.

BACON, Jason W. MIPS Instruction Code Formats. *In*: *Computer Science 315 Lecture Notes*. 2010. Disponível em: https://web.archive.org/web/20200707161825/www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch05s07.html. Acesso em: 19 jan. 2023.

BETTEREXPLAINED. *Understanding Big and Little Endian Byte Order*. c2023. Disponível em: https://betterexplained.com/articles/understanding-big-and-little-endian-byte-order/. Acesso em: 18 nov. 2022.

BLANC, Bertrand; MAARAOUI, Bob. *Endianness or Where is Byte 0?*. 2005. Disponível em: http://3bc.bertrand-blanc.com/endianness05.pdf. Acesso em: 18 nov. 2022.

BROWN, Andrew. Announcing wasi-threads.

Bytecode Alliance, 2023. Disponível em: https://bytecodealliance.org/articles/wasi-threads. Acesso em: 14 mar. 2023.

CAN I USE. *WebAssembly*. c2023. Disponível em: https://caniuse.com/wasm. Acesso em: 8 set. 2022.

CRICHTON, Alex. Multithreading Rust and Wasm. *Rust and WebAssembly*, 2018. Disponível em: https://rustwasm.github.io/2018/10/24/multithreading-rust-and-wasm.html. Acesso em: 20 ago. 2022.

GOHMAN, Dan. *WASI*: WebAssembly System Interface. 2019. Disponível em: https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-overview.md. Acesso em: 14 mar. 2023.

HAAS, Andreas; ROSSBERG, Andreas; SCHUFF, Derek L.; TITZER, Ben L.; HOLMAN, Michael; GOHMAN, Dan; WAGNER, Luke; ZAKAI, Alon; BASTIEN, JF. Bringing the Web Up to Speed with WebAssembly. *In*: *PLDI 2017*: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2017. p. 185–200. Disponível em: https://dl.acm.org/doi/pdf/10.1145/3062341.306236

3. Acesso em: 14 jul. 2023.

HAMILTON, Eric. What is Edge Computing: The Network Edge Explained. *Cloudwards*, 2018. Disponível em: https://www.cloudwards.net/what-is-edge-computing/. Acesso em: 13 maio 2023.

HEIKKINEN, Ilmari. Typed arrays - Binary data in the browser. *Web.dev*, 2012. Disponível em: https://web.dev/webgl-typed-arrays/. Acesso em: 16 ago. 2022.

HERLIHY, Maurice P.; WING, Jeannette M. Linearizability: A Correctness Condition for Concurrent Objects. *In*: *ACM Transactions on Programming Languages and Systems*, v. 12, n. 3, jul. 1990, p. 463-492. Disponível em: https://citeseerx.ist.psu.edu/doc/10.1.1.142.5315. Acesso em: 23 ago. 2022.

KERRISK, Michael. *The Linux Programming Interface*: A Linux and UNIX System Programming Handbook. San Francisco: No Starch Press, 2010. p. 118-119.

MOZILLA. *JavaScript interface*. c2022. Disponível em: https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface. Acesso

em: 17 ago. 2022.

MOZILLA. *Understanding WebAssembly text format*. c2022. Disponível em: https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format. Acesso em: 16 ago. 2022.

MOZILLA. *Using the WebAssembly JavaScript API*. c2022. Disponível em: https://developer.mozilla.org/en-US/docs/WebAssembly/Using_the_JavaScript_API. Acesso em: 16 ago. 2022.

MOZILLA. *WebAssembly Concepts*. c2022. Disponível em: https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts. Acesso em: 16 ago. 2022.

MOZILLA. *WebAssembly instruction reference*. c2022. Disponível em: https://developer.mozilla.org/en-US/docs/WebAssembly/Reference. Acesso em: 14 ago. 2022.

PIKE, Rob. Hello World or Καλημέρα κόσμε or こんにちは 世界. *USENIX Winter 1993 Conference*, San Diego, CA, 1993. Disponível em

https://www.cl.cam.ac.uk/~mgk25/ucs/UTF-8-Plan9-paper.pdf. Acesso em: 6 fev. 2023.

PIKE, Rob. *UTF-8 history*. 2003. Disponível em: https://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt. Acesso em: 6 fev. 2023.

ROSSBERG, Andreas. *WebAssembly Specification*. c2023. Disponível em: https://webassembly.github.io/spec/core/. Acesso em: 26 ago. 2022.

THE RUST REFERENCE. *Linkage*. c2022. Disponível em: https://doc.rust-lang.org/.reference/linkage.html. Acesso em: 22 set. 2022.

SHEDLETSKY, John J. Comment on the Sequential and Indeterminate Behavior of an End-Around-Carry Adder. *In*: *IEEE Transactions on Computers*, v. 26, n. 3, p. 271-272, mar. 1977. Disponível em: https://ieeexplore.ieee.org/document/1674817. Acesso em: 14 jul. 2023.

SMITH, James E.; NAIR, Ravi. The Architecture of Virtual Machines. *In*: *Computer*, v. 38, n. 5, p. 32-38, maio 2005. Disponível em:

https://minds.wisconsin.edu/bitstream/handle/1793/11154/file_1.pdf. Acesso em: 14 jul. 2023.

STEELE, Guy L. *Common Lisp the language*. 2. ed. Woburn: Digital Press, 1990.

STEPANYAN, Ingvar. Using WebAssembly threads from C, C++ and Rust. *Web.dev*, 2021. Disponível em: https://web.dev/webassembly-threads/. Acesso em: 20 ago. 2022.

TOAL, Ray. *Intermediate Representations*. Disponível em: https://cs.lmu.edu/~ray/notes/ir/. Acesso em: 10 fev. 2023.

UNICODE. *Unicode® 6.0.0*. c2023. Disponível em: https://www.unicode.org/versions/Unicode6.0.0/. Acesso em: 4 fev. 2023.

VARDA, Kenton. WebAssembly on Cloudflare Workers. *Cloudfare*, 2018. Disponível em: https://blog.cloudflare.com/webassembly-on-cloudflare-workers/. Acesso em: 13 maio 2023.

WALKER, David. Intermediate Representation. *In*: *Computer Science 320: Compilers*, 2003. Disponível em

https://www.cs.princeton.edu/courses/archive/spr03/cs320/notes/IR-trans1.pdf. Acesso em: 10 fev. 2023.

WASMTIME. *WASI Proposals Support*. c2023. Disponível em: https://docs.wasmtime.dev/stability-wasi-proposals-support.html. Acesso em: 13 mar. 2023.

WASMTIME. *WebAssembly Proposals Support*. c2023. Disponível em: https://docs.wasmtime.dev/stability-wasm-proposals-support.html. Acesso em: 14 mar. 2023.